12-2017

Scalable Digital Architecture of Hierarchical Temporal Memory Spatial Pooler

Sadhvi Praveen
sp2378@rit.edu

Scalable Digital Architecture of Hierarchical Temporal Memory Spatial Pooler

Sadhvi Praveen

Scalable Digital Architecture of Hierarchical Temporal Memory Spatial Pooler

Sadhvi Praveen

December 2017

A thesis submitted in partial fulfillment
of the requirements for the Degree of
Master of Science
in
Computer Engineering

Department of Computer Engineering
Kate Gleason College of Engineering

Rochester Institute of Technology
Rochester, New York
December 2017

Scalable Digital Architecture of Hierarchical Temporal Memory Spatial Pooler

Sadhvi Praveen

Committee Approval:

Dr. Dhireesha Kudithipudi *Advisor* Date
Professor

Dr. Marcin Lukowiak Date
Associate Professor

Dr. Sonia Lopez *Alarcon* Date
Associate Professor

To my loving family,

The driving force behind who I am today,

Thank you for believing in me and encouraging me to believe in myself.

Acknowledgments

I take this opportunity to express my sincere gratitude to my advisor Dr. Dhireesha Kudithipudi for her support, patience, and motivation. Besides my advisor, I would like to thank my thesis committee members: Dr. Marcin Lukowiak and Dr. Sonia Lopez Alarcon, for their insightful comments and hard questions.

I am also thankful for our research team at the NanoComputing Research Lab, in particular James Mnatzaganian and Lennard Streat, for their advice and feedback. Also, I thank Anvesh Polepalli and Tej Pandit for all the stimulating discussions and their cooperation.

I must express my profound gratitude to all the faculty members of the department of Computer Engineering at RIT for equipping me with the technical expertise needed to accomplish the task of completing this thesis.

Last but not least, I am greatly indebted to my family and friends for their continuous support and encouragement throughout my years of study, as this would not have been possible without them.

Abstract

Hierarchical Temporal memory is an unsupervised machine learning algorithm. Inspired by the structural and functional properties of the human brain, it is capable of processing spatio-temporal signals which are used for data storage and predictions. The algorithm is composed of two main components; the Spatial Pooler and the Temporal Memory. The spatial pooler produces a sparse distribution representation for the given pattern. These generalized representations are used by the temporal memory to make predictions. Therefore, it is important to ensure that more generalized sparse distribution representations are obtained for the spatio-temporal data patterns.

This work presents digital design of spatial pooler implementation for an existing mathematical algorithm along with analysis of its scalability for the target FPGA device. The digital design is implemented in two ways; Conventional and Parallel architectures. The architectures are compared in terms of speedup, area and power consumption. Based on the analysis of results, it is seen that the parallel approach is more efficient in terms of speed and power, with a negligible increase in device utilization. The spatial pooler design is evaluated against the standard MNIST dataset, obtaining upto 90% and 88% classification accuracy for the train and test data, respectively. Additionally, the designs are tested on the MNIST dataset, in the presence of noise, to determine its robustness. Fluctuations of upto 10% of the peak accuracy are observed during classification, and are noted in the classification accuracy plots for the dataset with noise. The design is synthesized for the Xilinx Virtex 7 family with a total power consumption of upto 260 mW.

Contents

Signature Sheet **i**

Dedication **ii**

Acknowledgments **iii**

Abstract **iv**

Table of Contents **v**

List of Figures **vii**

List of Tables **xiii**

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

1.3 Document Outline 3

2 Background **4**

2.1 Background 4

2.1.1 First Generation Algorithm 6

2.1.2 Second Generation Algorithm 6

2.1.3 Hierarchical Temporal Memory 7

2.1.4 Sparse Distributed Representation 8

2.1.5 HTM Spatial Pooler 9

3 Digital Architecture **16**

3.1 Design Methodology 16

3.1.1 HTM Spatial Pooler Module 17

3.1.2 Receiver Module 19

3.1.3 Initialization Module 20

3.1.4 Overlap Module 24

3.1.5 Inhibition Module 29

3.1.6 Learning Module 30

3.1.7 User Defined Parameters 33

3.2 Implementation 33
3.2.1 Conventional mHTM SP Architecture 33
3.2.2 Parallel mHTM SP Architecture 36

4 Results and Analysis **39**
4.1 Verification 39
4.1.1 MNIST Dataset 39
4.1.2 Module Verification 41
4.1.3 HTM SP Classification Accuracy 42
4.2 Simulations and Analysis 43
4.2.1 Synthesized Model Device Utilization 50

5 Conclusion **56**
5.1 Conclusion 56

6 Future Work **57**

List of Figures

2.1 High level representation of the Hierarchical Temporal Memory. All the regions are hierarchically stacked with the lowest level regions exposed directly to the information from the input domain. The upper level regions accept processed data from the lower level regions. The top most regions directly send out processed data/pattern information to the output domain. Each region consists of multiple columns with each column consisting of numerous cells within them. Columns and their cells are represented in one of the regions as shown. 5
2.2 Sparse Distributed Representation 8
2.3 Phase 1: The Overlap phase sets the overlap value to zero for each new column and counts the number of proximal synapses that are active and connected. The column overlap is determined by comparing the active connected synapses count with the segment threshold, seg th. In the flowchart, C represents the total number of columns and c is a single column in question. Similarly, S is the total number of synapses per column and s is the synapse in question. The operation is performed until all the synapses and columns are computed. Any column with overlap value lesser than the threshold will be kept from participating in the next phase. 11
2.4 Phase 2: The inhibition phase obtains the desired column activity for all the neighboring columns in the region. To mark a column as a winning column, the process checks if the column's overlap is greater than the desired column activity. The winning column participates in creating SDR for a given input data pattern. The procedure terminates after all the column overlap values are processed. 13
2.5 Phase 3: The Learning phase checks if a column is a winner. If it is a winning column, all the permanence values of the active proximal synapses are incremented by the Pinc constant value. If the proximal synapse of the winning column is not active, its permanence value is decremented by the pdec constant value. The permanence values of those columns that are not winners remain unchanged. 14

3.1 High level RTL representation of spatial pooler and its submodules. 17

3.2 RTL representation of receiver module. It consists of a counter to track the number of input bits fed in. A demux is used to generate address for the input bit. Upon assertion of start signal, the input bits are placed onto the bit position pointed to by the generated address. The resulting parallel data pattern is transmitted to the Spatial pooler. The module waits for the Ack signal from the Spatial pooler to accept the next set of serial input bits and the process repeats. 19

3.3 RTL representation of Initialization module showing submodules Permanence generator module and LFSR module. Initialization module starts when Start is asserted. This in turn starts the Permanence generator module. When the Permanence generator completes its execution, it produces ACK which triggers the Start line of the LFSR module. Initialization module produces ACK when LFSR module completes its execution. 20

3.4 RTL representation of 10-bit LFSR. It consists of serially connected D flip flops each being driven by the output of the previous. All the flip flops are driven at the rising edge of the same clock. The outputs from the flip flop 9 and flip flop 2 are fed into a two-input XOR gate and connected to the flip flop 0 for tapping. 21

3.5 RTL representation of LFSR Module. It consists of LFSR to generate random column indices. If the generated random number is lesser than the input pattern bit length, then the counter increments and the value is stored in the BRAM address generated by the counter. The values outside the range will be discarded. ACK is set high upon completion. 22

3.6 RTL representation of Permanence generator module. P-bit counter generates the memory address for BRAM. The value of P depends on the number of permanence values to be generated based on the number of columns in the spatial pooler. A 2-bit counter is used for select lines for the decoder and multiplexer. The counters are incremented every clock cycle. Adder 1 and two's compliment adder 1 generate the permanence values between the range minimum permanence value and synapse threshold. Adder 2 and two's compliment adder 2 generate the permanence values between the range synapse threshold and maximum permanence value. Upon generation of all the permanence values, ACK is set high. 23

3.7 Phase 1: RTL design of Accumulating sub module of the Overlap module. When the start is triggered, the synapse counter and the accumulator values are set to zero. The module fetches the permanence and input bit values from their respective BRAMs. The module computes the sum of all the active and connected proximal synapses in a column. When all the proximal synapses of the column are computed, ACK is produced. 25

3.8 Phase 1: RTL design of the Overlap module, along with Accumulator and comparing sub modules. The unshaded region represents the comparing module. When the reset is high, the column counter is set to zero. C represents columns and the value of C depends on the number of columns in the spatial pooler. Upon start, the overlap module starts executing the Accumulator module and waits for the ACK from Accumulator module. The ACK from the Accumulator module triggers the start line of the comparing module. The comparing module compares the sum value from the Accumulator module with segment threshold. The column counter is incremented and ACK is produced. ACK triggers the overlap module and it also triggers the Accumulator module. The counter counts till the total number of columns and produces Output done to indicate completion of overlap process for the given input pattern. 27

3.9 Phase 2: RTL representation of Inhibition module. C represents the number of columns in the spatial pooler and K is the percentage of winning columns. The column overlap is sent to all the comparators to sort the overlap values in descending order. The column index values of the topmost K overlap values are stored in the Bit index register. When all the column overlap values are sorted, the column index values in the bit index register are used to set SDR bits. The values in the bit index represents the position of the bits in the SDR that needs to be active. 28

3.10 Phase 3: RTL design of Learning module. The values c update and pdec are user defined constant parameter values. SDR output produced by Inhibition value is used as the input and is assigned to the signal y. C-Bit counter counts up to number of columns in the spatial pooler and it is set to zero when the reset is high. y(counter) represents the 'counter'th position bit in signal y. The permanence and column indices values are accessed from the BRAMs using their addresses. The module computes the equation for the Learning phase [2.5]. During calculation of the equation, overflow is taken care of by appending an extra bit on the MSB. The appended MSB bit is discarded when the final updated permanence value for the synapse of a column is obtained. 31
3.11 Representation of the execution flow of spatial pooler and its lower level modules in Conventional mHTM SP implementation. 34
3.12 Representation of the execution flow of spatial pooler and its lower level modules in Parallel mHTM SP implementation. 37

4.1 Samples of MNIST dataset and their labels. [12] 40
4.2 State machine represents training and testing phases of the digital design. The spatial pooler remains idle until the start signal is asserted and the spatial pooler training begins. Learning remains enabled throughout and all the train data images are fed in to learn the images. When all the train images are processed, training is complete and the learning is disabled. Now the spatial pooler is tested for test data images and the process continues until all the test data images are fed in. 42
4.3 Graphical representation of design verification flow of the spatial pooler. MNIST data is processed to perform image thresholding. After the image process, the data is sent to the spatial pooler module to obtain its SDR. The spatial pooler output is sent to support vector machine to obtain the classification accuracy of the spatial pooler using MNIST label for the input data. 43
4.4 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 100 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10. 44

4.5 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 200 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10. 45
4.6 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 400 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10. 45
4.7 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 784 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10. 46
4.8 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 100 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30. 47
4.9 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 200 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30. 47
4.10 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 400 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30. 48
4.11 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 784 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30. 48
4.12 Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a regions with 100, 200, 400 and 784 columns and 100 proximal synapses per column, keeping their parameter values fixed at their optimum values. 49

4.13 Plots showing classification accuracy of the spatial pooler design for MNIST train dataset with noise. 1 represents the performance for spatial pooler with 100 columns; 2 represents the performance for spatial pooler with 200 columns; 3 represents the performance for spatial pooler with 400 columns; 4 represents the performance for spatial pooler with 784 columns. 51
4.14 Plots showing classification accuracy of the spatial pooler design for MNIST test dataset with noise. 1 represents the performance for spatial pooler with 100 columns; 2 represents the performance for spatial pooler with 200 columns; 3 represents the performance for spatial pooler with 400 columns; 4 represents the performance for spatial pooler with 784 columns. 51
4.15 Plots comparing the execution time for the Convention mHTM SP and Parallel mHTM SP architectures and their speedup. 53
4.16 Plots comparing total power consumption for Conventional and Parallel mHTM SP architectures on Xilinx Virtex7 XC7VX330T target device. 1 indicates power consumption for the spatial pooler with 100 columns. 2 indicates power consumption for the spatial pooler with 200 columns. 3 indicates power consumption for the spatial pooler with 400 columns. 54
4.17 Plots comparing individual component power consumption on for the Conventional mHTM SP and Parallel mHTM SP Architectures with 400 columns and 100 synapses per column on Virtex 7 FPGAs 55

List of Tables

4.1 User defined constant vales for the spatial pooler parameters. 50
4.2 Area estimates of Conventional mHTM SP architecture on a Xilinx Virtex7 XC7VX330T target device. 52
4.3 Area estimates of Parallel mHTM SP architecture on a Xilinx Virtex7 XC7VX330T target device. 52
4.4 Execution timing comparison for the Conventional and Parallel mHTM SP architectures on a Xilinx Virtex7 XC7VX330T target device. . . . 53
4.5 Power estimates of Conventional and Parallel mHTM SP architectures on a Xilinx Virtex7 XC7VX330T target device. 54

Chapter 1

Introduction

1.1 Motivation

In everyday life, most information we encounter is represented in time and space dimensions. The human brain is capable of sensing spatio-temporal data and processing it. Our brain performs cognitive tasks like visual pattern recognition without consuming much power. These tasks are processed in the neocortex, which is considered to be the seat of intelligence in the brain. Even though powerful computers have emerged that are capable of performing many tasks better than humans, the cognitive functions that seem easy to us, may be extremely difficult for computers. Additionally, the power consumption is also significantly higher.

To build an intelligent machine that can operate the way the brain does, a design that is capable of processing the spatio-temporal patterns is needed. One such theory was proposed in the year 2003, known as Hierarchical Temporal Memory (HTM). This algorithm enables online learning of the spatio-temporal sequences and can make predictions based on their learned sequences. The algorithmic definition of HTM is still evolving. Unlike other algorithms, HTM is not problem specific. It learns by exposing itself to a stream of data consisting of a sequence of patterns. The

performance capability of the HTM algorithm is dependent on the type of input dataset it is trained on.

For an algorithm to process data the way the brain does, it is necessary for the algorithm to mimic the structural and functional characteristics of the neocortex of the brain. HTM is said to have emulated the properties of the neocortex both structurally and functionally. Therefore, it is said to be have the capability of performing complex tasks like object recognition, image recognition and prediction and, hence, has its applications in the field of machine learning.

Hardware implementation of HTM will explore the area and power requirements for the algorithm on hardware. Since the HTM algorithm is evolving, it becomes important to choose a re-configurable hardware platform in order to incorporate any future changes made in the algorithm. Field Programmable Gate Array (FPGA) is one such technology that has thousands of configurable logic blocks (CLBs) and allows reprogramming of the design when required. Therefore, the implementation of HTM on FPGAs will be less expensive making the research and development cost effective.

1.2 Objectives

The aim of this work is to implement the mathematical model of the Hierarchical Temporal Memory Spatial Pooler (mHTM SP) [1] on an FPGA. With this intent, the following tasks were performed:

- Implemented two different architectures for mHTM SP and compared their speedup, area and power requirements.

- Developed a code in MATLAB to perform image processing of the input MNIST dataset.
- Evaluated the RTL design for the Spatial Pooler against the MNIST dataset using a Support Vector Machine classifier in Python.

1.3 Document Outline

This thesis work is documented as follows:

Chapter 2 provides the background and previous works for the Hierarchical Temporal Memory algorithm. The evolution of the generations of the algorithm is explained in brief. The chapter contains the description of the structure of the Cortical Learning Algorithm Hierarchical Temporal Memory, algorithmic description, and mathematical model of the Spatial Pooler in detail.

Chapter 3.1 describes the approach taken for RTL implementation of the mathematical formalization of the Spatial Pooler.

Chapter 4 mentions the design verification, the Spatial Pooler output evaluation, and the dataset that the Spatial Pooler is exposed to in this work. This chapter also discusses the experimental results for the synthesized RTL of the Spatial Pooler. Results of simulations such as accuracy and synthesis area information may be obtained here.

In chapter 5, summary of the different improvements and strategic modifications made in this hardware implementation of the Spatial Pooler are listed.

Chapter 6 outlines the various methodologies which can be implemented to further the scope of this research. It prescribes several suggestions and directions for any research conducted as an addendum to this thesis.

Chapter 2

Background

Hierarchical temporal memory (HTM) is a machine learning algorithm originally proposed by Jeff Hawkins[8] in 2003. The algorithm is inspired by the neocortex of the mammalian brain and follows memory prediction framework[1]. It aims at solving pattern recognition problems by emulating the structural and functional properties of the neocortex in the algorithm. This memory based system has the capability to store patterns and their sequences. Given a spatio-temporal data, HTM learns the input sequences and makes predictions based on the learned patterns.

2.1 Background

Hierarchical Temporal Memory, the theoretical framework for the neocortex, follows a brain function theory that hypothesizes the pattern storage and memory utilization in the neocortex. To capture the key structural properties of the neocortex, the HTM consists of regions. The term "hierarchical" in the name stands for multiple regions that are hierarchically stacked in the form of a tree. These regions process the cortical information. The input spatio-temporal patterns from the external input domain are directly connected to the lowest regions in the tree and the highest region of the HTM is exposed to the output domain. These regions are responsible for

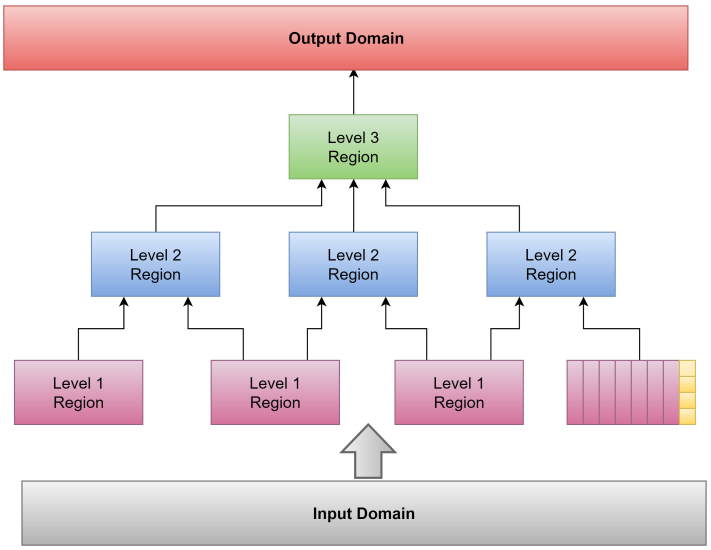

Figure 2.1: High level representation of the Hierarchical Temporal Memory. All the regions are hierarchically stacked with the lowest level regions exposed directly to the information from the input domain. The upper level regions accept processed data from the lower level regions. The top most regions directly send out processed data/pattern information to the output domain. Each region consists of multiple columns with each column consisting of numerous cells within them. Columns and their cells are represented in one of the regions as shown.

input patterns processing, learned representations storage and their predictions. The algorithm follows a bottom-up approach with inputs being fed at the bottom and output obtained at the top. Since the lowest regions are directly exposed to the input, more detailed information of the patterns is stored in these regions. The stored patterns from these regions are successively used as inputs to the upper regions in the hierarchy, that are connected to these regions, and the pattern information becomes less detailed as it flows up. The pattern information is converged in the higher regions and represents the HTM output and it is obtained at the highest region. The hierarchical nature of the algorithm reduces the training time of the network and its memory usage.

Two generations of algorithm were proposed for the HTM, namely; Zeta and

Cortical learning algorithm. Following sections 2.1.1 and 2.1.2 explain briefly about the two generations.

2.1.1 First Generation Algorithm

The first-generation algorithm for HTM was called Zeta[9] which was designed and published in the year 2007 by Dileep George. The work presented that HTM Zeta was an unsupervised algorithm and utilized the Bayesian inference and Markov graphs for pattern recognition[13]. The algorithm accepted time varying data as its input and followed offline learning process. The algorithm consisted of Zeta nodes which are considered as the fundamental working units of the network for Zeta. The nodes are hierarchically placed in the shape of a tree. Multiple nodes in the tree constituted a region. The nodes at the lower level receive the data from the input and pass the information on to the higher-level nodes that are connected to it. The information obtained from the bottom nodes are converged to obtain the representation of the data objects. The nodes were responsible for learning the patterns and the learned pattern representations storage. The complete discussion of the working principles of the algorithm is found in [14].

HTM Zeta is no longer supported and is replaced by the second-generation algorithm of HTM, Cortical learning algorithm.

2.1.2 Second Generation Algorithm

The second-generation algorithm for HTM is called Cortical learning algorithm (CLA). This algorithm is an improved version of the first-generation algorithm. This algorithm was released in the official white paper[10] in 2011 and is popularly known as Hierarchical Temporal Memory Cortical learning algorithm (HTM CLA). The HTM

CLA emulates the neocortex of the brain which is responsible for performing cognitive tasks. The nodes from the HTM Zeta algorithm are replaced by columns in HTM CLA. Each column in turn consist of a number of cells. The cells within the columns are the basic functional unit for HTM CLA. HTM CLA focuses more on Sparse distribution representations (SDR)[2.1.4] making it significantly different from the HTM Zeta. SDR is used as it is hypothesized that the brain stores its information in the form of SDR. Also, using SDRs for the patterns enables reduction in the complexity of the algorithm. Numenta CLA whitepaper suggests that HTM CLA consists of two components; Spatial Pooler (SP) and Temporal Pooler (TP).

Research in the field of HTM has led to implementations of CLA on both software and hardware domains. Zhou and Luo[16] developed a many-core implementation of HTM and obtained parallelism at the columnar level. Price[15] developed CLA for multi-core architectures using OpenMP. He showed that algorithm's scalability could be achieved by aggressive optimizations. Zyrah et al[17] implemented a hardware architecture modeling HTM CLA. The work resulted in a significant speedup.

Latest version of the HTM CLA is released and is referred to as just Hierarchical Temporal Memory (HTM). This version is similar to the HTM CLA with a slight difference, in which the Temporal Pooler of HTM CLA is referred to Temporal Memory (TM) in HTM.

2.1.3 Hierarchical Temporal Memory

HTM has two distinct components namely Spatial Pooler (SP) and Temporal Memory (TM). The spatial pooler is responsible for creating the Sparse Distributed Representations (SDRs) for the given spatio-temporal input data patterns. The spatial pooler

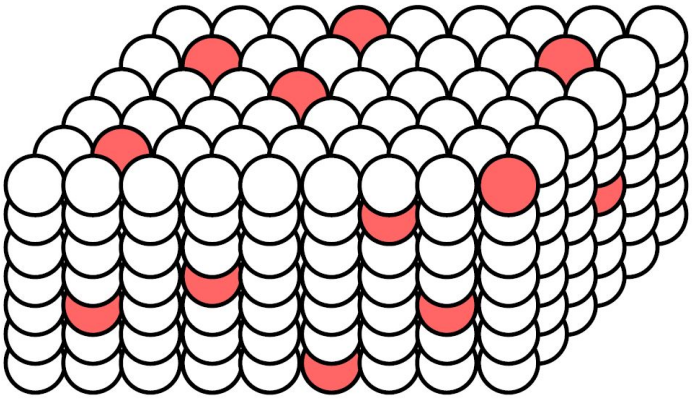

Figure 2.2: Sparse Distributed Representation

algorithm follows unsupervised learning and uses vector quantization to perform its operations. The temporal memory of the HTM, on the other hand, is responsible for making predictions using the learned sequences for the given input. The working principle of temporal memory component is similar to Hebb's rule. Learning an input sequence involves formation of connections between cells within the active columns in the region.

2.1.4 Sparse Distributed Representation

Sparse Distribution Representation (SDR) is an encoding technique in which only a small percentage of the overall bits are active at a time. HTM works based on the principles of SDR, and the spatial pooler is responsible to convert the HTM input patterns into SDRs. Regardless of how many input bits are active, the spatial pooler region converts the given input to an internal representation with only a small percentage of the bits active at a time. This encoding process could be lossy but the loss of information is often negligible since the loss will not have a substantial effect. The usage of the SDRs also reduces the memory needed to store the data and the power consumption in the hardware.

2.1.5 HTM Spatial Pooler

The HTM spatial pooler algorithm follows an unsupervised, iterative, and online learning process. It uses its previously learned knowledge of the pattern sequences to make any modifications for the given pattern. Spatial pooler can be viewed as a mapping function that accepts inputs from the input domain and maps them to their respective patterns in the feature domain in the form of SDR. All the SDRs in the feature domain are the generalized representations of the patterns from the input domain.

The HTM spatial pooler region consists of columns. Each column in the region has numerous proximal synapses. The columns are connected to the input space and to each other through the proximal synapses via proximal segments. The degree of connectivity of these synapses are determined by the value of their weights, called permanence. A synapse is connected if its permanence value is greater than a threshold value, known as the synapse threshold, otherwise; unconnected. The synapse is active if an active input is given to it, otherwise; inactive. The permanence values of the synapses range between 0 and 1, inclusive. Closer the value of the permanence is to value 1, the more connected the synapse is. The synapses with values closer to value 0 are likely to be not connected. The input patterns are randomly connected to the spatial pooler columns. The spatial pooler operates at the column level and hence columns are the working units for the spatial pooler.

Mnatzaganian et al[1] proposed the mathematical framework for the HTM CLA spatial pooler. The framework provides a probabilistic approach for the initialization and a better understanding of the algorithm for hardware implementation. Streat et al[18] implemented a scalable hardware realization of HTM CLA spatial pooler

using large scale solid state flash memory. The design was validated in TSMC 180nm process.

The following sections explain the algorithmic definition of HTM spatial pooler. The spatial pooler operation involves three phases namely; Overlap, Inhibition, and Learning.

2.1.5.1 Initialization

When the spatial pooler operation begins, the network is initialized to prepare it for its operation on initial input patterns. The spatial pooler columns are connected to random subset of input pattern bits through proximal synapses. The permanence values of the proximal synapses for the columns in the region are initialized randomly after the formation of connections.

The permanence values initialized are random and are closer to the synapse threshold value to ensure that the synapse is connected in the first few training iterations. The permanence values are initialized such that half of the synapses in a column are connected and the other half are almost closer to being connected increasing the chances of all the columns to participate in computing the input patterns.

The mathematical representation of the initialization process is given by the equation 2.1[1]. Uniform distribution in the equation ensures that at least half of the proximal synapse permanence values are connected and the initialized values range between a small window size of a user defined constant value ϕ_d.

$$\mathbf{\Phi}_{i,k} \sim Unif\big(\rho_s - \phi_d, \rho_s + \phi_d\big) \tag{2.1}$$

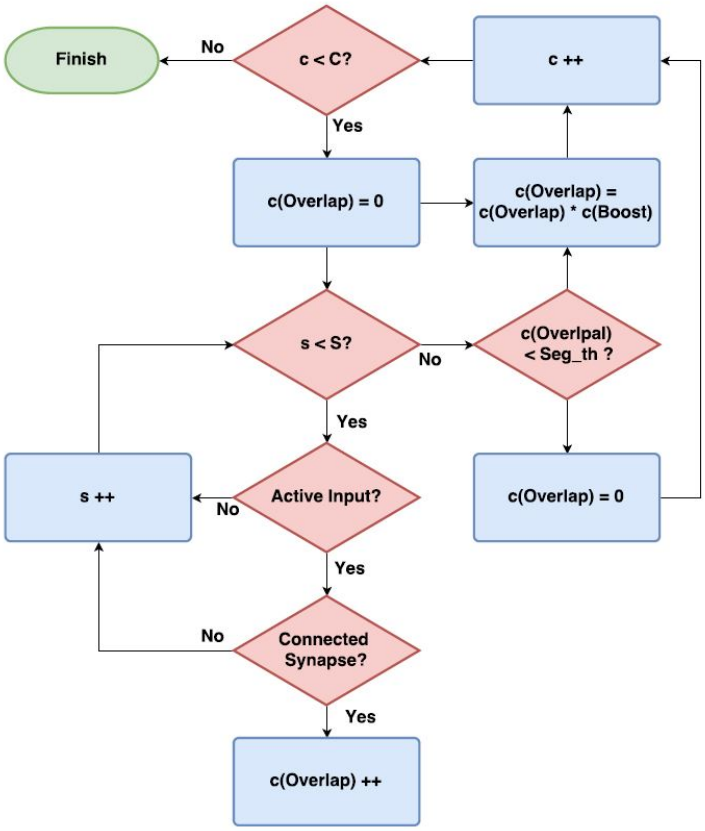

Figure 2.3: Phase 1: The Overlap phase sets the overlap value to zero for each new column and counts the number of proximal synapses that are active and connected. The column overlap is determined by comparing the active connected synapses count with the segment threshold, seg th. In the flowchart, C represents the total number of columns and c is a single column in question. Similarly, S is the total number of synapses per column and s is the synapse in question. The operation is performed until all the synapses and columns are computed. Any column with overlap value lesser than the threshold will be kept from participating in the next phase.

2.1.5.2 Phase 1: Overlap

For every new input pattern, the spatial pooler begins by computing the overlap values for all the columns in the region. The degree of connectivity of individual synapses for all the columns is obtained by calculating the total number of synapses that are active and connected in each column. The overlap of the column is required to determine its activation in the next phase. The overlap is computed by comparing the total number of active connected synapses in the column with the proximal segment

threshold value. If the number of active and connected synapses in the column is greater than or equal to the threshold value, the total value times the column boost constant value is assigned as the overlap value for that column, and the column will be activated in the next phase. Otherwise, the column's overlap value is set to zero and hence the column is inactive in the next phase.

Multiplying the column overlap with its respective boost constant is called Boosting. This technique increases the column's chances of participation in the next phase. It also penalizes the columns that are not active frequently. Boosting is optional and may be disabled based on users requirements.

Mathematical representation of overall operation of Phase 1 is given by equation 2.2[1]. The result is a vector of overlap values for all the columns which is represented by $\hat{\alpha}_i$.

$$\vec{\alpha} \equiv \begin{cases} \hat{\alpha}_i \hat{\beta}_i & \hat{\alpha}_i \geq \rho_d, \quad \forall i \\ 0 & otherwise \end{cases} \tag{2.2}$$

2.1.5.3 Phase 2: Inhibition

The second phase of the HTM spatial pooler involves determining the winning columns that participate in generating the generalized SDRs for the given input data patterns. To achieve sparsity of active bits in the spatial pooler output, only a small percentage of the columns will be marked as winners. ρ_c number of columns with largest overlap values will be the winning columns. The parameter ρ_c is a user defined constant called desired column activity. This parameter determines the percentage of active bits in the SDR output. SDR produced for the given input pattern is the output for

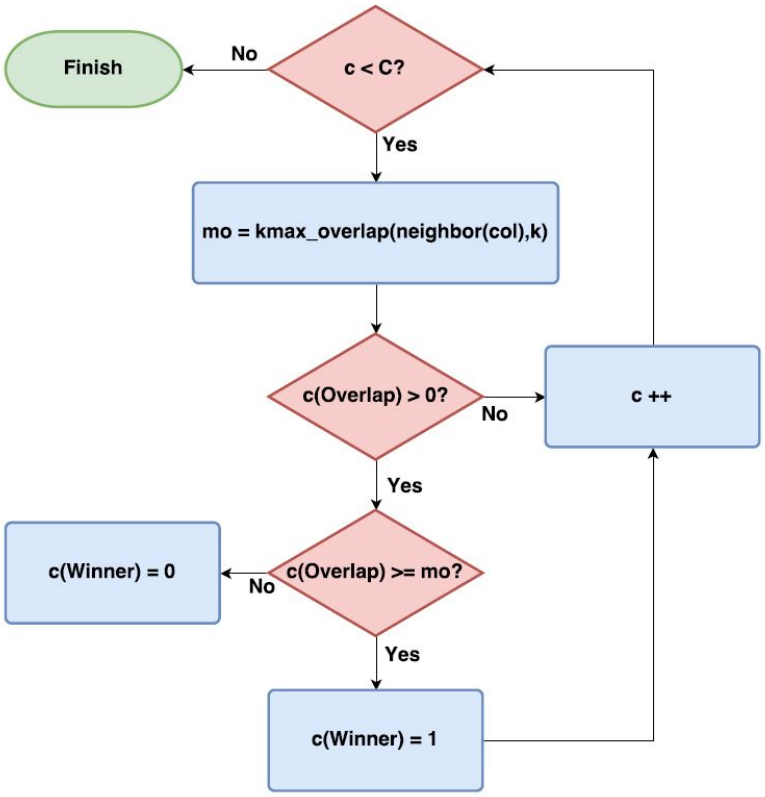

Figure 2.4: Phase 2: The inhibition phase obtains the desired column activity for all the neighboring columns in the region. To mark a column as a winning column, the process checks if the column's overlap is greater than the desired column activity. The winning column participates in creating SDR for a given input data pattern. The procedure terminates after all the column overlap values are processed.

the SP.

The mathematical representation of the phase 2 of the SP algorithm is as shown in the equation 2.3. The value of γ in the equation is calculated using equation 2.4. At first, the $\rho_c th$ largest overlaps are computed using function kmax. The max function returns the maximum value and ensures that the value obtained is bound to 1 and above only.

$$\vec{\hat{c}} \equiv I\left(\vec{\alpha_i} \geq \vec{\gamma_i}\right) \forall i \tag{2.3}$$

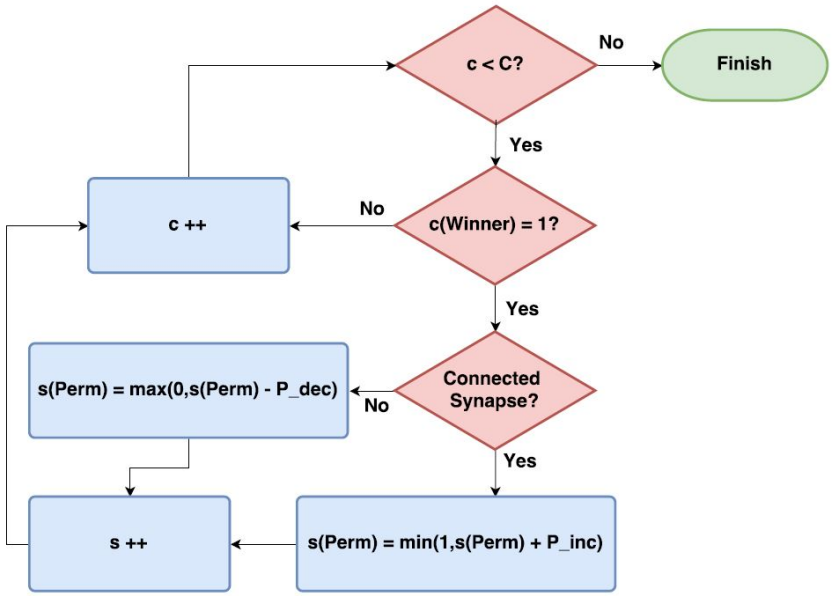

Figure 2.5: Phase 3: The Learning phase checks if a column is a winner. If it is a winning column, all the permanence values of the active proximal synapses are incremented by the Pinc constant value. If the proximal synapse of the winning column is not active, its permanence value is decremented by the pdec constant value. The permanence values of those columns that are not winners remain unchanged.

$$\vec{\gamma} \equiv max\big(kmax\big(\boldsymbol{H_i} \odot \vec{\boldsymbol{\alpha}}, \rho_c\big), 1\big) \forall i \tag{2.4}$$

2.1.5.4 Phase 3: Learning

The third phase of the HTM spatial pooler is called the Learning phase and it involves computation of the updated permanence values for the proximal synapses of all the winning columns. During the learning phase, the permanence values of the active proximal synapses in the winning columns are incremented by a permanence increment constant value and if the proximal synapse is inactive, the permanence values are decremented by a permanence decrement constant value. The range of

permanence values is bound between values 0 and 1, inclusive.

The learning phase also updates the boosting factors for all the columns and inhibition radius. The boost value for each column is determined by duty cycles; namely, active duty cycle and overlap duty cycle. Learning is optional and is generally enabled while training the HTM spatial pooler, and disabled while testing.

Mathematically, learning phase of the SP algorithm is defined as per equation 2.5.

$$\boldsymbol{\Phi} \equiv clip\big(\boldsymbol{\Phi} \oplus \boldsymbol{\delta\Phi}, 0, 1\big) \tag{2.5}$$

$$\boldsymbol{\delta\Phi} \equiv \vec{\hat{c}}^{T} \odot \big(\phi_{+}\boldsymbol{X} - \big(\phi_{-}\neg\boldsymbol{X}\big)\big) \tag{2.6}$$

The clip function in the equation makes sure that the updated permanence values are bound between 0 and 1, inclusive. $\delta\Phi$ is the permanence update value which is used to increment or decrement the current permanence value during learning process. This permanence update value $\delta\Phi$ is calculated using equation 2.6.

Chapter 3

Digital Architecture

This chapter explains the approach taken to realize the mathematical model of the Hierarchical Temporal Memory's Spatial pooler on an FPGA in detail. The reconfigurable property of FPGA supports scalability of mHTM SP[1] algorithm when implementated on a board.

3.1 Design Methodology

The digital design of Spatial pooler for FPGA consists of four modules – Initialization module, Overlap module, Inhibition module and Learning module. The Register-Transfer Level (RTL) design of the spatial pooler is implemented on Virtex7 FPGA using VHDL (Very High Speed Integrated Circuit Hardware Descriptive Language). For hardware efficiency, all the fractional values of the parameters of the mHTM SP [1] are replaced by the integer values in the design. This reduces the number of bits required to represent the values, which in turn reduces the area required for designing the overall system architecture.

The following sections explain all the blocks of the architecture in detail.

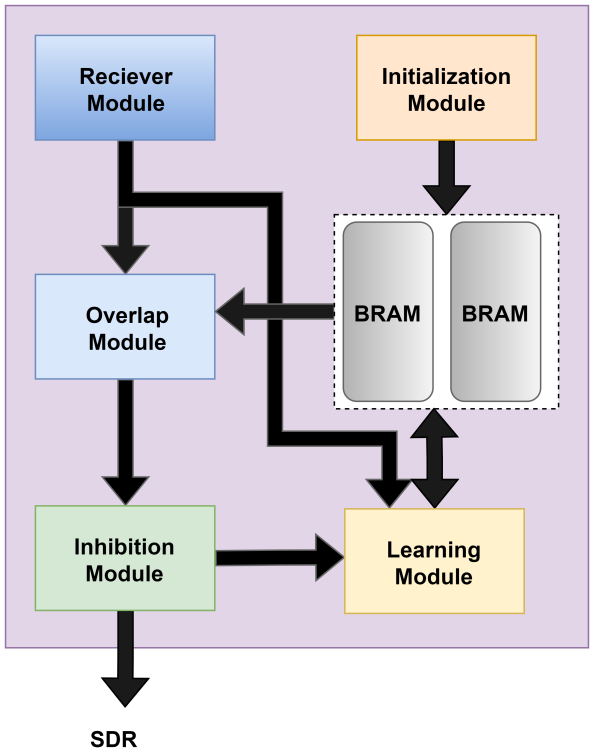

Figure 3.1: High level RTL representation of spatial pooler and its submodules.

3.1.1 HTM Spatial Pooler Module

This is a top-level module for the spatial pooler. It consists of a datapath and a control unit. The function of this module is to accept the input patterns from the receiver, connect all the lower level modules, control the flow of execution of modules, enable and disable the network training and testing processes, and produce the spatial pooler SDR output to the feature domain or external source. Moore Finite State Machine (FSM) is used throughout the design.

The datapath assigns values to all the signals in their respective state and the control unit controls switching of states by constantly monitoring the given conditions, if any. All the modules are synchronous and are designed to work on the rising edge of the clock. The modules consist of active high clock enable which helps in reducing

the overall power consumption of the Spatial pooler.

The modules follow handshaking technique, so when acknowledge is produced from the lower level modules, the HTM SP module enables next module and disables the current module, in the next clock cycle. The HTM SP module consists of registers to hold operating parameters, like the number of iterations performed, to determine when the spatial pooler is trained and when it is ready for test.

The HTM SP module begins with its operation when the start input is asserted. Before accepting the patterns, the HTM SP module instantiates the initialization module to produce the required initial values. After the initialization, the HTM SP module accepts the input pattern from the receiver module. The initialization module produces an acknowledge after the completion of execution for the given input pattern indicating the spatial pooler is ready to accept input data patterns. The data patterns are processed by the HTM SP module. This procedure is iterative and the HTM SP module learns on every iteration when the learning parameter is enabled. An iteration is complete after an acknowledge from the Learning module is produced, in case of training process. During testing process, an iteration is complete after an acknowledge from Inhibition module is produced.

The way HTM SP module accepts the input patterns remains the same way for both training and testing processes. The module must wait for a longer execution time to start next iterative pattern during training since the training involves three phases - Overlap, Inhibition, and Learning. Testing requires comparatively lesser execution time since the process involves only Overlap and Inhibition phases. The total number of iterations for both training and testing is defined by the user in the user defined package file[3.1.7].

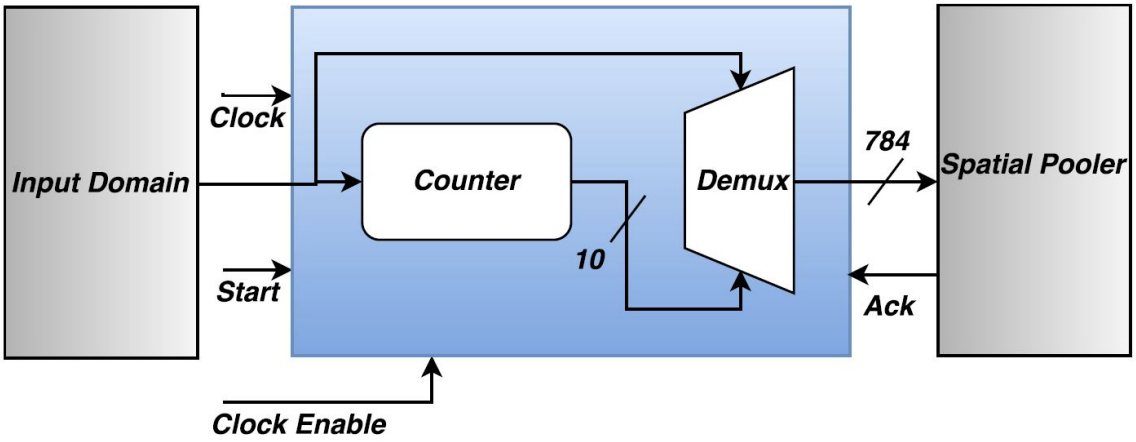

Figure 3.2: RTL representation of receiver module. It consists of a counter to track the number of input bits fed in. A demux is used to generate address for the input bit. Upon assertion of start signal, the input bits are placed onto the bit position pointed to by the generated address. The resulting parallel data pattern is transmitted to the Spatial pooler. The module waits for the Ack signal from the Spatial pooler to accept the next set of serial input bits and the process repeats.

3.1.2 Receiver Module

Receiver module accepts the input patterns serially, converts the incoming serial data to parallel for the HTM SP module. The memory address from which the input bit is accessed in the HTM SP module is obtained by the values produced from the LFSR module. The input pattern is presented bit-wise to the input ports of the HTM SP module, starting from the LSB up to MSB. The input bit counter in the receiver module monitors the number of bits sent to the HTM SP module. The length of the input pattern is declared in the user defined package file and can be varied based on the input dataset chosen. The SP process begins when all the pattern bits are sent to the HTM SP module. To accept the next input pattern, the receiver waits till the HTM SP sends an acknowledge indicating completion of the spatial pooler process.

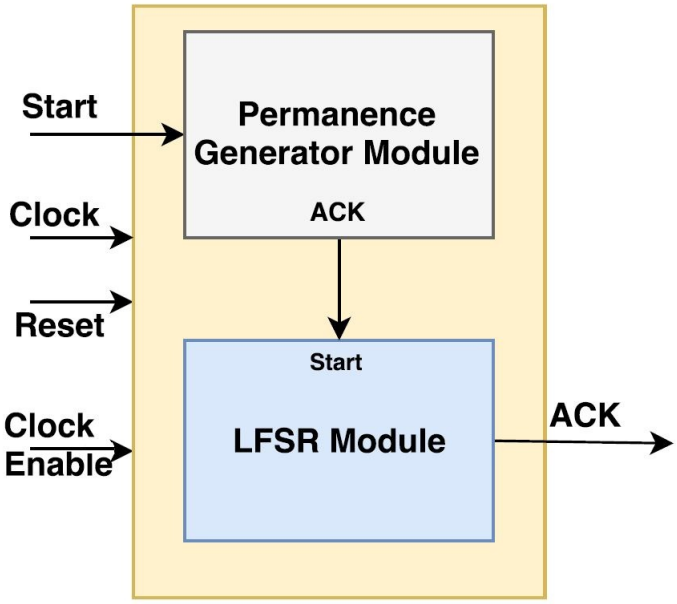

Figure 3.3: RTL representation of Initialization module showing submodules Permanence generator module and LFSR module. Initialization module starts when Start is asserted. This in turn starts the Permanence generator module. When the Permanence generator completes its execution, it produces ACK which triggers the Start line of the LFSR module. Initialization module produces ACK when LFSR module completes its execution.

3.1.3 Initialization Module

Upon instantiation, this module is the first to commence operations. It executes only once, at the beginning, to produce the initial values and is not reused unless the system needs to be reset entirely. The module mainly consists of Linear Feedback Shift Register (LFSR) and Permanence Generator sub-modules.

The module is responsible for producing the initial values for the proximal synapses for the spatial pooler columns and the column indices for the input pattern bits. The generated column indices and the proximal synapse values are stored in separate Random Access Memory (RAM) blocks and are accessed using their respective memory addresses. The size of BRAM for the column indices is equal to the length of input data pattern and the size remains the same throughout the design process. The size

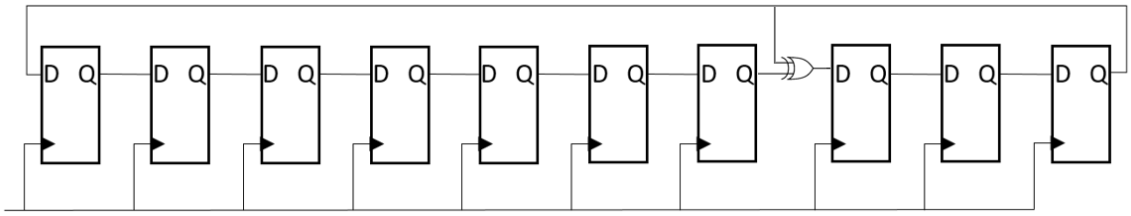

Figure 3.4: RTL representation of 10-bit LFSR. It consists of serially connected D flip flops each being driven by the output of the previous. All the flip flops are driven at the rising edge of the same clock. The outputs from the flip flop 9 and flip flop 2 are fed into a two-input XOR gate and connected to the flip flop 0 for tapping.

of BRAM for the proximal synapses depends on the number of spatial pooler columns and vary as the digital design of the spatial pooler is designed to support scalability. The following sections 3.1.3.1 and 3.1.3.2 explain the techniques used for the generation of random column indices for the input bit connection and the permanence values for the synapses in all the columns in the RTL design.

3.1.3.1 Linear Feedback Shift Register Module

LSFR is a pseudo-random number generator (PRNG) and the numbers generated are used as the elements of the set of input indices for all the patterns. The column's proximal synapses are connected to these input bits in a random fashion. The LFSR module consists of M number of D-flip-flops connected serially and 2-input XOR gates for tapping. The value of M is determined by the length of the input pattern for the SP. All the D flip flops operate at the same clock edge. The number of two input XOR gates used in the LFSR are flexible. The module is designed to generate random values in the range 1 to total number of bits in the input pattern, inclusive. The range of random numbers can be varied by adding or removing D-flip-flops in the LFSR module.

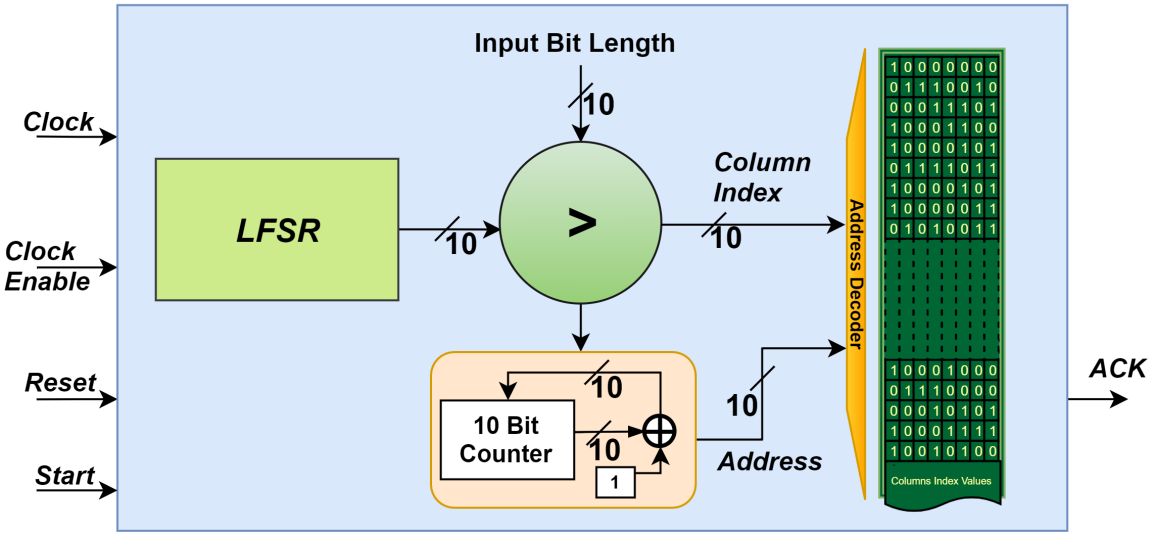

Figure 3.5: RTL representation of LFSR Module. It consists of LFSR to generate random column indices. If the generated random number is lesser than the input pattern bit length, then the counter increments and the value is stored in the BRAM address generated by the counter. The values outside the range will be discarded. ACK is set high upon completion.

In this work, a 10-bit LFSR is designed since the length of the input pattern given to the spatial pooler is 784. For simplicity and resource optimization purpose, a single two input XOR gate is used for tapping. The values from 1 to 784 are produced in a random order and transmitted to the HTM SP module for storage in the memory for the input pattern data access. Any generated value outside of the range is ignored by the HTM SP module. Therefore, there will be no out of the range values or repetitions of the numbers in the memory.

PRNGs require a seed value before they can generate a random sequence of numbers. The LFSR module has an active-high asynchronous reset and when it is high, a 10-bit arbitrary seed value is assigned to the LFSR. The design then waits for the assertion of LFSR signal to start with the random number generation and continues to generate the numbers until 784 random values are generated and then the LFSR

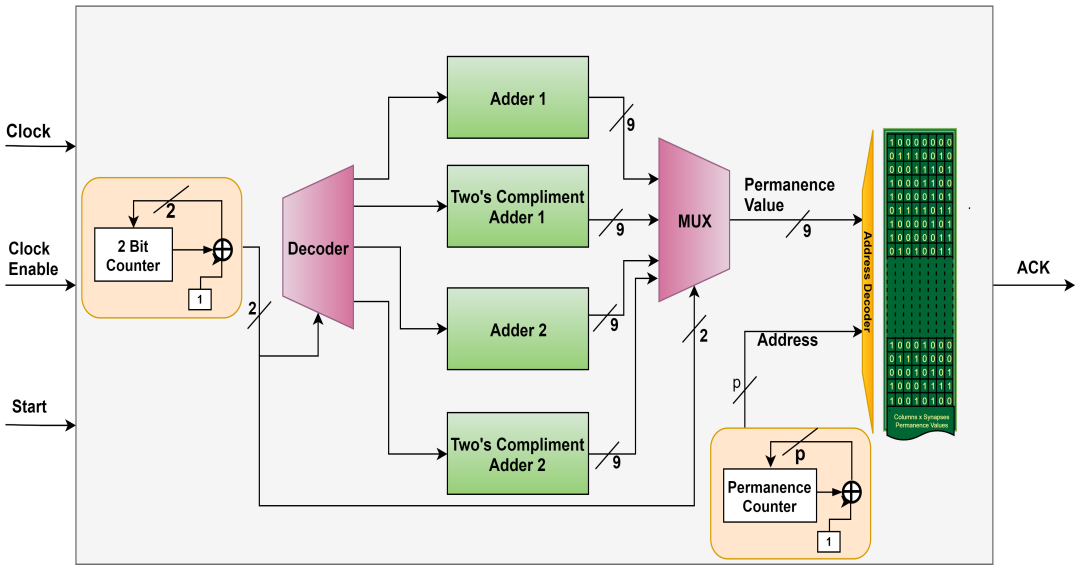

Figure 3.6: RTL representation of Permanence generator module. P-bit counter generates the memory address for BRAM. The value of P depends on the number of permanence values to be generated based on the number of columns in the spatial pooler. A 2-bit counter is used for select lines for the decoder and multiplexer. The counters are incremented every clock cycle. Adder 1 and two's compliment adder 1 generate the permanence values between the range minimum permanence value and synapse threshold. Adder 2 and two's compliment adder 2 generate the permanence values between the range synapse threshold and maximum permanence value. Upon generation of all the permanence values, ACK is set high.

is disabled.

3.1.3.2 Permanence Generator Module

A set of permanence values for the proximal synapses for all the columns are generated using adders and two's complement adders, whose function is equivalent to the equation 2.1. The module is designed to generate permanence values within a small range closer to synapse threshold value to ensure fair selection of each column in the architecture. At least half of the proximal synapse values generated for each column will be connected.

The initialization module waits for the active high enable to start generating the

values. When the permanence generator module is instantiated, the adders and the two's compliment adders are initialized with the synapse threshold, maximum and minimum permanence constant values. The maximum and the minimum permanence values are user defined constant values and they define the range of the permanence values that can be generated. This range can be varied, based on the range requirement, by changing maximum and the minimum permanence values in the package file.

When the module is enabled, every clock cycle one of the adders increment the synapse threshold value until the maximum permanence value is reached and the other adder increments the minimum permanence value until the synapse threshold is reached. Similarly, one of the two's compliment adders decrements the synapse threshold value until the minimum permanence value is reached and the other decrements maximum permanence value until the synapse threshold is reached. When the above-mentioned condition is reached, the module reloads the adders with their respective initial values, with which they started the calculation at the beginning and the process repeats. This results in generation of equal number of connected and unconnected values in a distributed manner. A counter in the HTM SP module keeps track of the number of permanence values generated and the permanence generator module is enabled until all the values are generated. The values generated are stored in the memory.

3.1.4 Overlap Module

After the HTM SP module accepts values from the receiver, overlap module is executed first for every new data iteration, in both training and testing processes of the

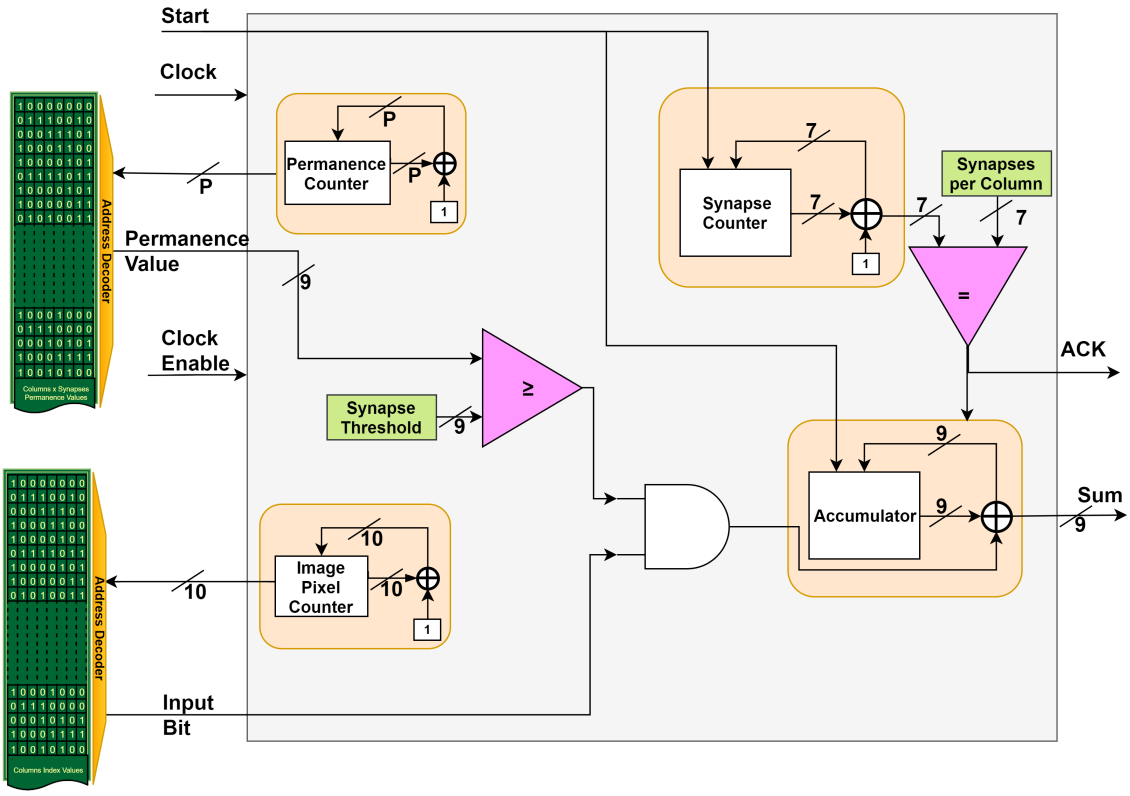

Figure 3.7: Phase 1: RTL design of Accumulating sub module of the Overlap module. When the start is triggered, the synapse counter and the accumulator values are set to zero. The module fetches the permanence and input bit values from their respective BRAMs. The module computes the sum of all the active and connected proximal synapses in a column. When all the proximal synapses of the column are computed, ACK is produced.

spatial pooler.

This module is responsible to access random input bits and permanence values from the Block RAMs and compute overlap value for each column based on the number of active connected synapses in the column. The Overlap module is divided into two sub modules; Accumulating and Comparing modules. The execution of the accumulating module is followed by the comparing module.

The accumulating module consists of an accumulator and an adder. It counts the total number of synapses that are active and connected in a column. This module

starts its execution for every new spatial pooler column and the execution continues until all the synapses of the column are counted. Upon completion, an acknowledge is produced. The overlap module waits for the acknowledge from the accumulating module to enable the comparing module and disable the accumulating module. The comparing module compares the resulting value from the accumulating module with the proximal segment threshold.

The above-mentioned procedure is repeated until all the columns are computed. The HTM SP module consists of a counter to monitor the number of columns processed.

Upon instantiation of the accumulating module, the value in the accumulator is initialized to zero. Accumulator is used to hold the active and connected synapses count in a column. Whenever an active input bit is detected and the proximal synapse is connected, the adder increments the value in the accumulator. When all the synapses of the column are computed, an acknowledge is asserted indicating completion of all the synapses in the column and the accumulator value is presented to the output line of the accumulating module.

The comparing module waits for the acknowledge from the accumulating module, upon assertion it accepts the accumulator value and the comparator compares it with the segment threshold value. The segment threshold constant is defined in the package file by the user and can be varied based on the requirement. If the accumulator value is lesser than the segment threshold, the comparing module presents zero to its output, otherwise, the accumulator value is presented to the output of the comparing module. The resulting value is the overlap for the column in question.

Column overlap values of all the columns produced by the overlap module are sent

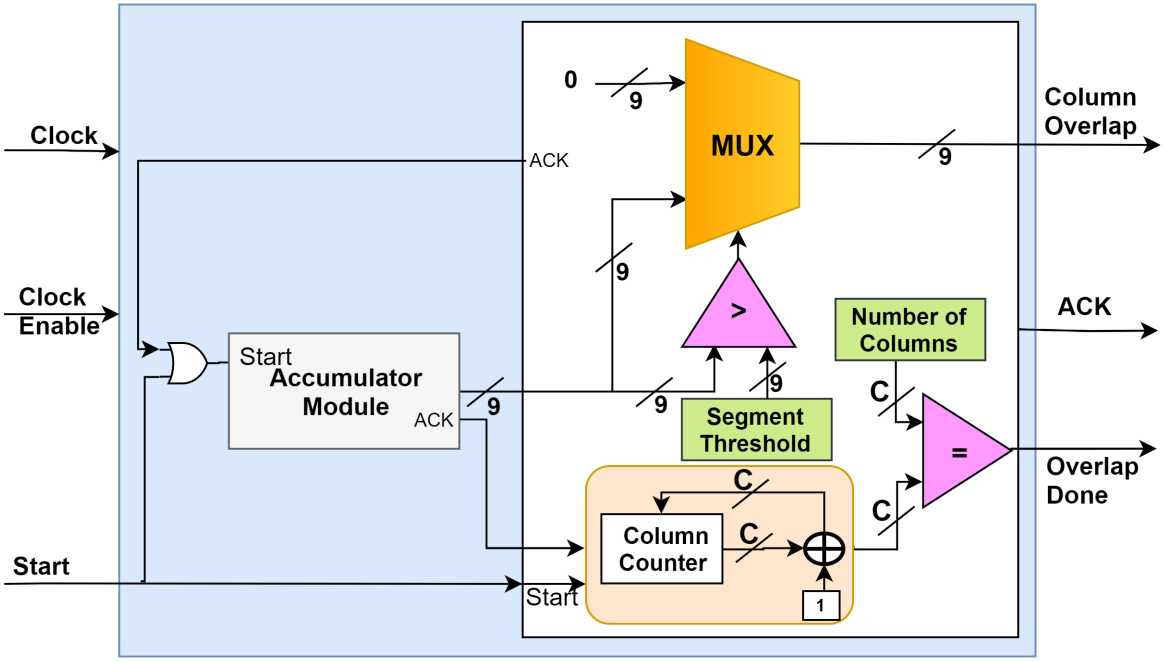

Figure 3.8: Phase 1: RTL design of the Overlap module, along with Accumulator and comparing sub modules. The unshaded region represents the comparing module. When the reset is high, the column counter is set to zero. C represents columns and the value of C depends on the number of columns in the spatial pooler. Upon start, the overlap module starts executing the Accumulator module and waits for the ACK from Accumulator module. The ACK from the Accumulator module triggers the start line of the comparing module. The comparing module compares the sum value from the Accumulator module with segment threshold. The column counter is incremented and ACK is produced. ACK triggers the overlap module and it also triggers the Accumulator module. The counter counts till the total number of columns and produces Output done to indicate completion of overlap process for the given input pattern.

to the Inhibition module as input.

For hardware simplification, column boosting technique is not implemented in the design. Therefore, boost value is kept constant at value 1 throughout. The functionality of Overlap module is equivalent to the equation 2.2. Bit-wise logical AND is performed in place of matrix multiplication for hardware optimization purpose.

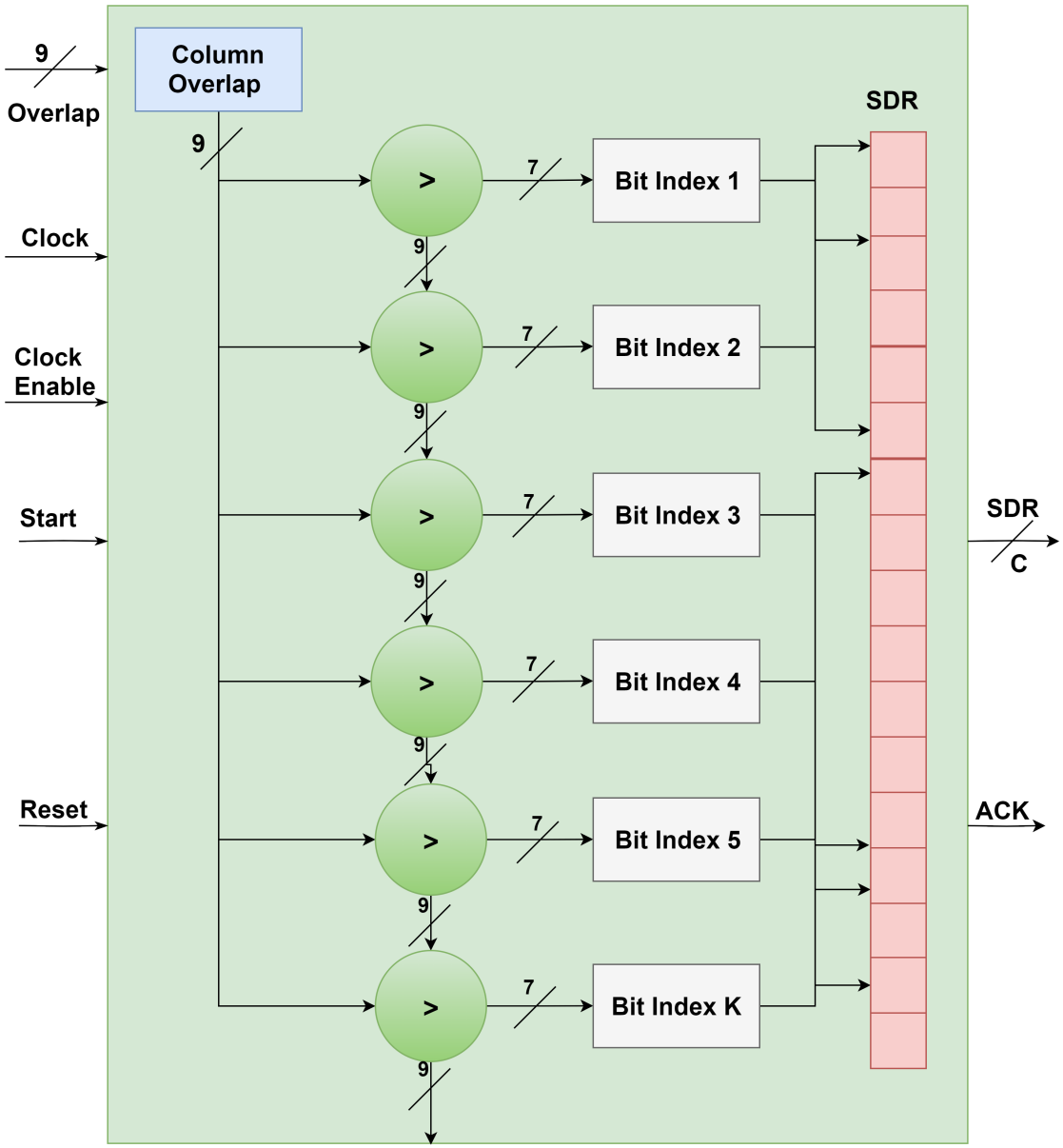

Figure 3.9: Phase 2: RTL representation of Inhibition module. C represents the number of columns in the spatial pooler and K is the percentage of winning columns. The column overlap is sent to all the comparators to sort the overlap values in descending order. The column index values of the topmost K overlap values are stored in the Bit index register. When all the column overlap values are sorted, the column index values in the bit index register are used to set SDR bits. The values in the bit index represents the position of the bits in the SDR that needs to be active.

3.1.5 Inhibition Module

The primary function of this module is to accept the column overlap values and compute the indices of winner columns in the spatial pooler. The output produced by this module is the generalized SDR for the given input and, hence, the output of the HTM SP module. The length of the SDR is equal to the number of columns in the spatial pooler and only up to K% of the total number of columns are made as winning columns in this design. The value of K is a user defined constant indicating the percentage of active bits in the SDR.

The module consists of a comparator and two vector arrays of length K. Bubble sort is performed on the column overlap values. Vector arrays are used to hold the values of top K overlap values and their respective index values after the sort.

Whenever a new column with overlap value greater than the previous overlap values in the vector is detected, the column with the least overlap value and its index in the vector arrays are removed. All the overlap values that are lesser than the new column overlap and their respective indices are shifted right by one place and the new column overlap is inserted.

The array with inhibition values is used to compare and keep track of the latest winning columns, while the array with column indices is used to compute the SDR representation of the input. This operation is equivalent to the equation 2.4.

At the beginning of the module execution, all the bits in the SDR are set to zero. When the overlap values of all the columns are sorted, the vector array with index values is accessed to determine the winning columns. Only those bits in the SDR pointed to by the index values are set to '1'. Once the SDR is computed, control signal allows the HTM SP Module to transmit the SDR for the given input

data pattern as the spatial pooler output. The functionality of Inhibition module is equivalent to the equation 2.3. The SDR generated by this module influences the update of permanence value of proximal synapses, therefore it is also sent to the Learning module as one of its input parameters.

Global inhibition is used in this design. The inhibition radius of the columns remains constant and is large enough to encapsulate all the columns. This yields higher computational efficiency, requires fewer parameters to be tracked, eliminating the need for dynamic update of inhibition parameters, saving hardware resources.

3.1.6 Learning Module

The execution of this module is enabled only during the training process of the spatial pooler, and is disabled during the test. Figure 3.11 represents the execution flow of HTM SP module during training and testing. The primary function of this module is to access the SDR for the given input pattern from the Inhibition module and update the permanence values of the proximal synapses of the winning columns only. The permanence values of proximal synapses of the remaining columns that are not qualified as winners remain unchanged.

The module basically consists of an adder and a two's complement adder, a comparator and few registers to hold the intermediate values, in order perform the function equivalent to the equation 2.5.

The learning module waits for the assertion of enable to start with the computation of the permanence update values. It continues to execute until all the columns in the SP region are computed. A counter in the HTM SP module keeps track of the number of columns and synapses being computed.

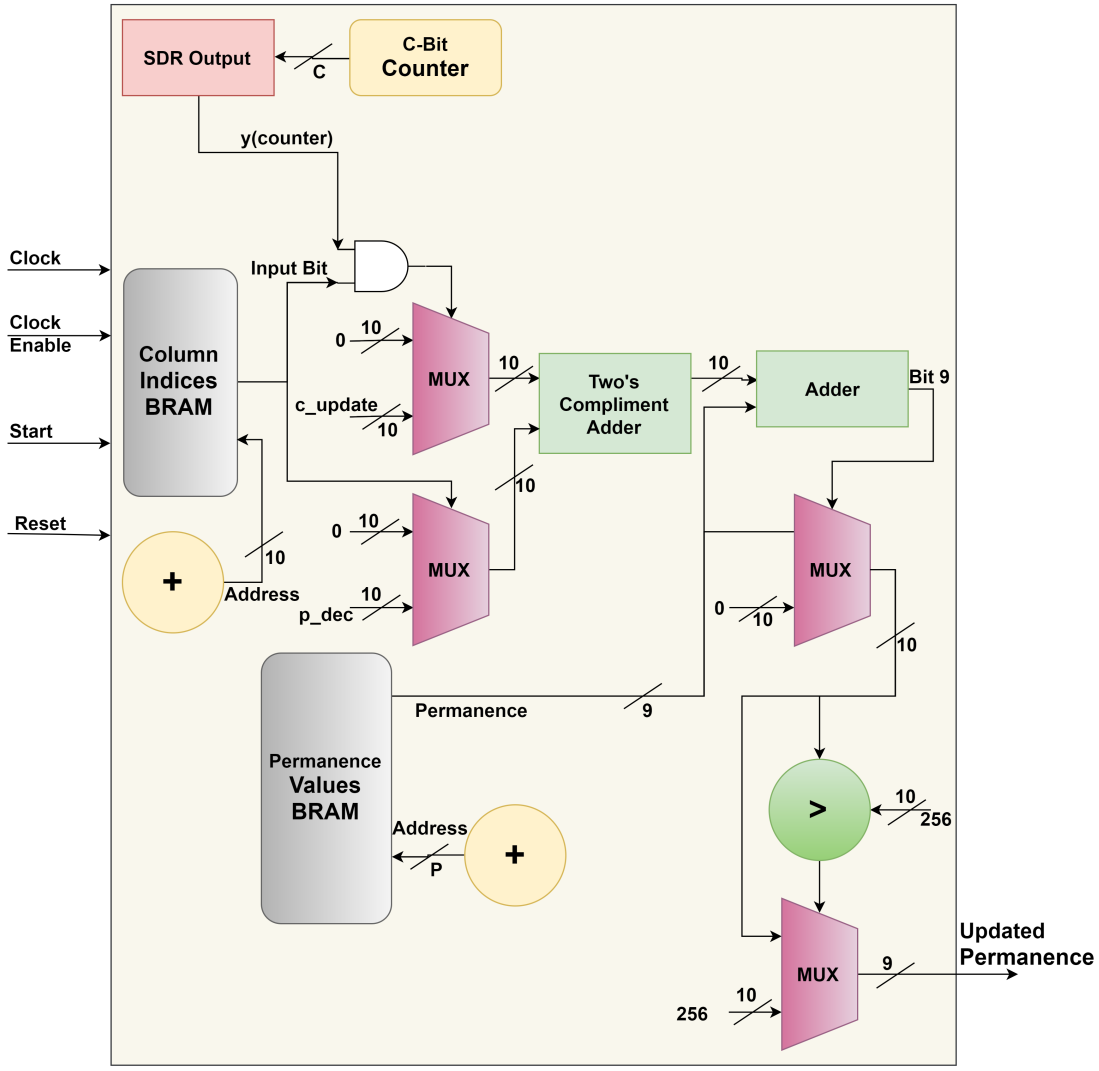

Figure 3.10: Phase 3: RTL design of Learning module. The values c update and pdec are user defined constant parameter values. SDR output produced by Inhibition value is used as the input and is assigned to the signal y. C-Bit counter counts up to number of columns in the spatial pooler and it is set to zero when the reset is high. y(counter) represents the 'counter'th position bit in signal y. The permanence and column indices values are accessed from the BRAMs using their addresses. The module computes the equation for the Learning phase [2.5]. During calculation of the equation, overflow is taken care of by appending an extra bit on the MSB. The appended MSB bit is discarded when the final updated permanence value for the synapse of a column is obtained.

To begin with, minuend and subtrahend for the subtraction in the equation 2.6 is calculated. The value of the minuend is dependent on the input bit and the column. If the column is a winner and the input bit is active, the user defined constants permanence increment and permanence decrement values are added and the total is assigned as the minuend. If either input bit is not active or the column is not a winner, the value of the minuend is set to zero. The value of the subtrahend is dependent on the column alone. If the column is a winner, permanence decrement value is assigned as the subtrahend value, otherwise, subtrahend is set to zero. The permanence update value is obtained by calculating the difference between the minuend and subtrahend values. Two's complement adder is used to perform subtraction of both the values. The operation is equivalent to equation 2.6.

The updated permanence values for the synapses are obtained by adding the permanence values of the synapse with the resulting value from subtraction. The total value obtained must be within the range 0 and 1, inclusive. Since the integer values are used to represent the fractional values of the mHTM SP [1], the permanence value produced by the learning module ranges between 0 and 256, with values in between representing fractions. To ensure the updated permanence value lies within the above-mentioned range, two comparators are used to perform clipping operation. One of the comparators clips the lower range while the other clips the upper range. The operation performed is equivalent to equation 2.5.

The permanence values of all the synapses of each columns in the BRAM are replaced with learned permanence values and are used in the next iteration.

All the multiplication operations from the equation 2.5 are replaced by bit-wise logical AND operations, thus reducing the utilization of the hardware resources.

3.1.7 User Defined Parameters

All the user defined parameters are declared in a separate VHDL package file. The spatial pooler design is scalable, therefore, the number of columns, proximal synapses per column, and hardware requirement can be varied by changing their respective values in the package file. The package file also contains user defined constant values like proximal synapse threshold, segment threshold and desired local activity. The package file is accessible to the HTM SP module and its submodules.

3.2 Implementation

In this work, two different design approaches are followed to implement the digital architecture for the mHTM SP algorithm; They are 1) conventional mHTM SP architecture and 2) Parallel mHTM SP architecture. Both the approaches make use of all the modules described in the section 3.1 to implement the mathematical model of the spatial pooler[1]. The functional results of both the implementations remains the same. They are different from each other from architectural implementation point of view. Following sections explain in detail the architectural implementations of the mHTM SP[1].

3.2.1 Conventional mHTM SP Architecture

This architecture is implemented with an intent to mimic the execution flow of mHTM SP framework [1] on the hardware. Initially all the modules for the spatial pooler described in section 3.1 remain idle. After the initial values for the spatial pooler are produced by the initialization module, the architecture accepts parallel input patterns from the receiver module. The execution of initialization module is followed by the

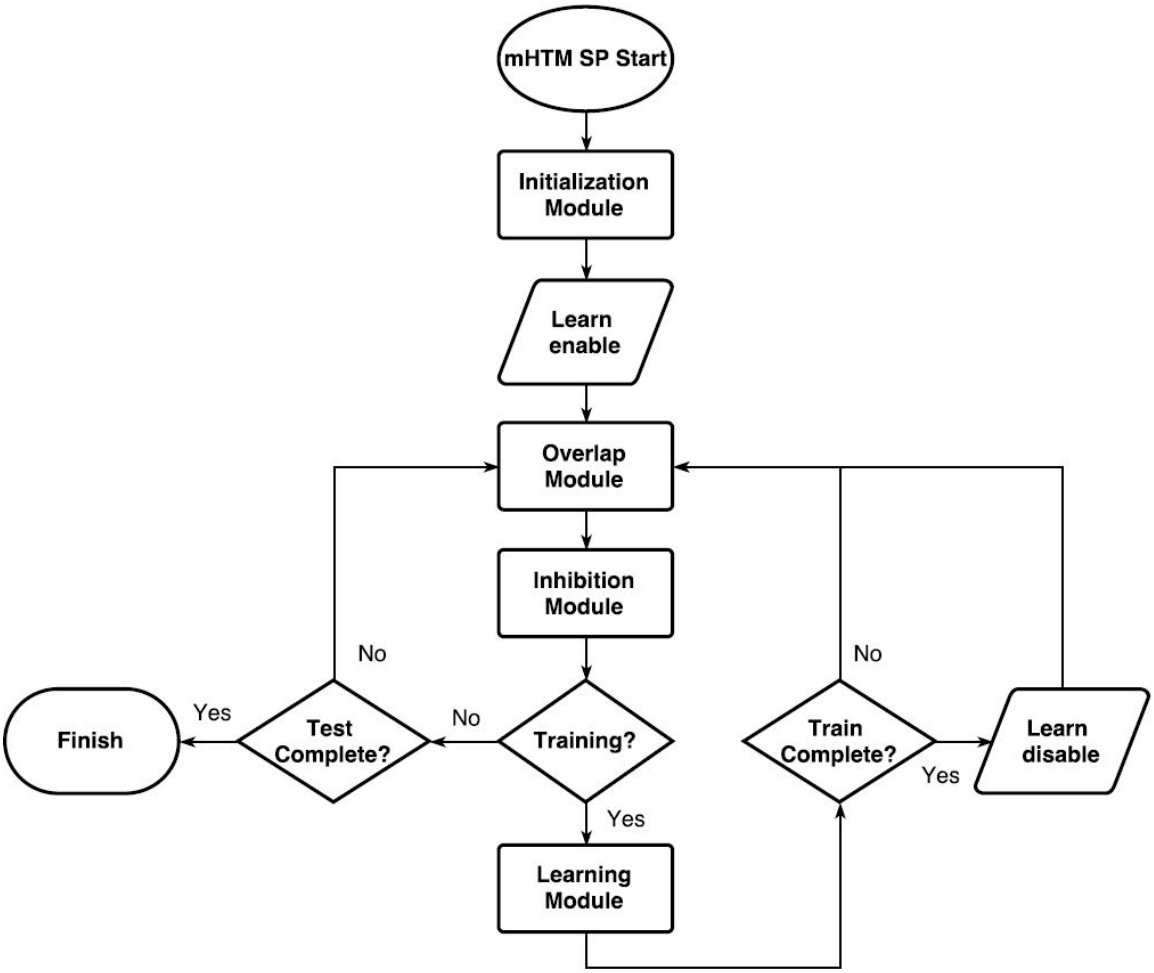

Figure 3.11: Representation of the execution flow of spatial pooler and its lower level modules in Conventional mHTM SP implementation.

overlap module.

All the columns and their synapses are processed sequentially throughout the design. When the synapses of the first spatial pooler column are processed, only the overlap module will remain in busy state and the other modules remain idle. Once the overlap for the first column is obtained, the overlap value is presented to the input line of the Inhibition module and the module continues with processing values of the synapses from the next column. At this point, the Inhibition module starts with the sorting process upon receiving column overlap value. The overlap and Inhibition

modules continue running in parallel hereafter. The Learning module remains idle until all the column overlap values are sorted and the SDR for the input pattern in question is produced. While the overlap and inhibition modules can run in parallel, the execution within each module remains sequential. The SDR for the given input is generated after all the columns are processed.

If network learning is enabled, the spatial pooler switches both overlap and inhibition modules to idle and the learning module is switched to busy state. The learning module continues to execute until all the permanence values are processed. During this time, the synapse permanence values are fetched from the memory sequentially, the updated permanence values for the synapses are calculated and written back. This operation requires two memory reads to fetch a permanence value from the BRAM, first read occurs during the overlap phase and the second read occurs during the learning phase. A memory write operation is performed to replace the updated permanence value back in the BRAM. Similarly, two memory read operations are performed to fetch the column indices from the BRAM.

If network learning is disabled, the learning module remains idle throughout, while the overlap and inhibition modules continue working as described. This operation requires a single memory read operation to fetch a permanence value from the BRAM, and, a memory read operation is performed to fetch the column indices from the BRAM.

The process is iterative and it continues to execute until all the input data patterns are fed in the spatial pooler. Figure 3.11 describes the pictorial representation of the modular execution flow for the architecture.

3.2.2 Parallel mHTM SP Architecture

This architecture is a modification of the Conventional mHTM SP Architecture[3.2.1] and provides an improved efficiency and speedup of the spatial pooler on hardware. In the previous design implementation 3.2.1, the execution of the learning module required a dedicated time, during which all the other modules remained idle. In this architecture design, learning module is computed along with the overlap module thereby reducing the overall execution time required.

The process begins with the execution of initialization module which is followed by accepting data from the receiver. All the values within the modules are computed serially. When the first input data pattern is given, for the synapses of the first column, only the overlap module will be busy and the other modules remain idle. Once the overlap is calculated, the overlap value is presented to the Inhibition module and the overlap module continues to process synapses for the next column. Inhibition and overlap modules continue running in parallel after the first column is processed. The Learning module remains idle until all the column overlap values are sorted and the SDR for the input pattern is obtained.

If network learning is enabled, the current input data pattern is moved to a separate register and the next set of input data pattern is accepted by the spatial pooler, before the execution of the learning module is enabled. The learning module uses the input data pattern stored in the separate register to compute the learned permanence values for that data pattern. During this time, the spatial pooler starts computing column overlap values for the newly accepted input data pattern.

Since both the overlap and learning modules use same parameters and are running in parallel, the fetched operands can be shared by both the modules. The

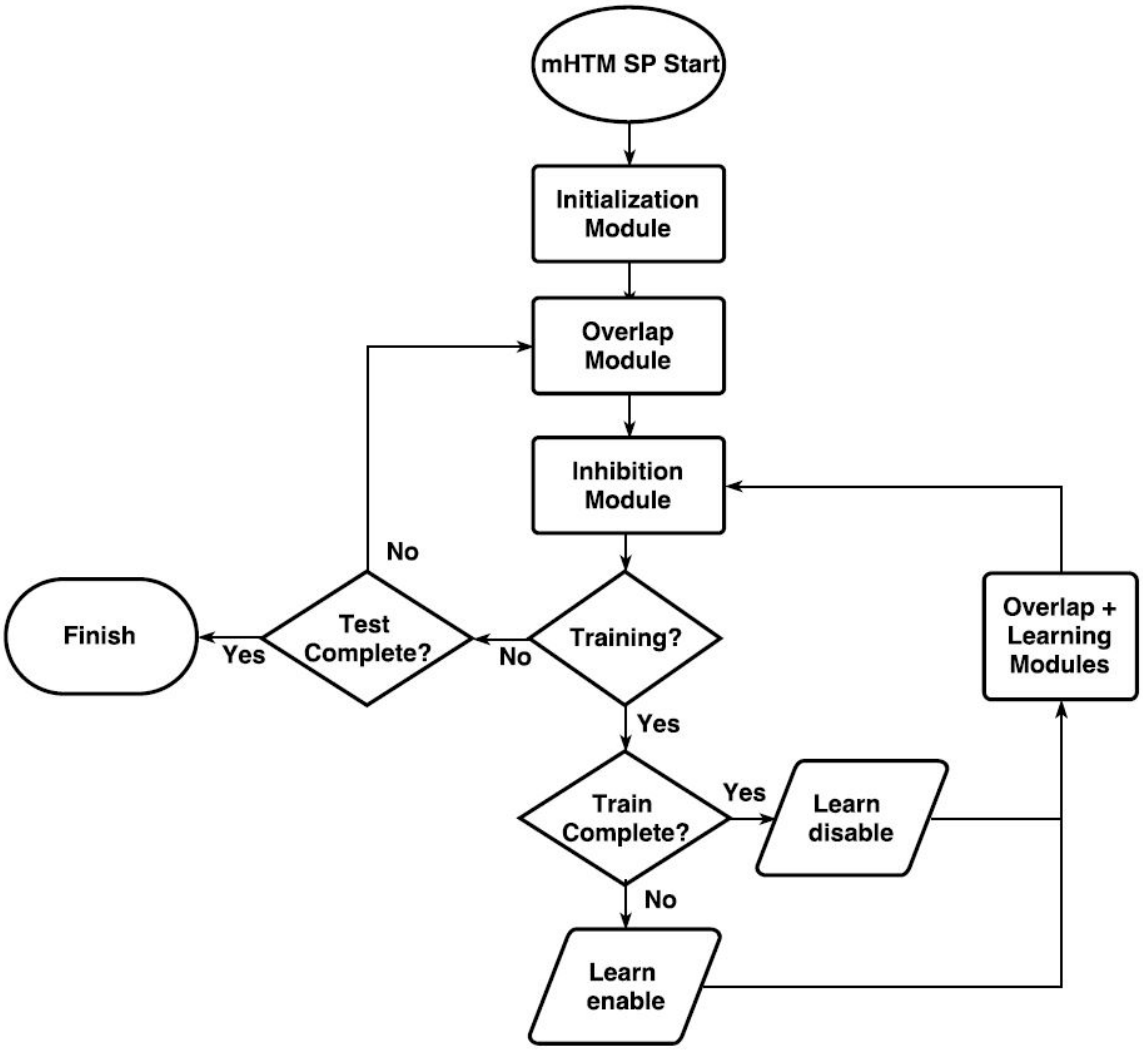

Figure 3.12: Representation of the execution flow of spatial pooler and its lower level modules in Parallel mHTM SP implementation.

architecture requires one memory read operation to fetch a permanence value from the BRAM, and, a memory write operation to write the updated permanence value back in BRAM. Similarly, one memory read operation for fetching the column indices from the BRAM. Therefore, the number of times the operands are read from the BRAM is reduced by half. The updated permanence value from the learning module is directly fed into the overlap module as the input permanence value for the next data pattern.

If network learning is disabled, the overlap and inhibition modules continue processing and the learning module remains idle throughout. Operand fetch involves single memory read operation for the permanence values in BRAM, and, one memory read operation is performed to fetch the column indices from the BRAM, for each data pattern.

The process is iterative and continues until all the input data patterns are fed into the spatial pooler. Figure 3.12 describes the pictorial representation of the modular execution flow for the architecture.

Chapter 4

Results and Analysis

The chapter presents the simulation results and analysis of spatial pooler architectures on the FPGA. The speedup, area and power requirements for the architectures are compared.

4.1 Verification

This section describes the techniques followed to verify the performance of the digital designs of the mathematical model for spatial pooler on the Modified National Institute of Standards and Technology (MNIST) database, with detailed descriptions of the input pre-processing, RTL design verification and the classification accuracy.

4.1.1 MNIST Dataset

Modified National Institute of Standards and Technology (MNIST) database of handwritten digits ranging from zero to nine is used to validate the performance of the RTL design of the spatial pooler algorithm. This standard database is composed of a set of 60,000 training and 10,000 testing images and their labels. Each image is a 28 x 28 pixel grayscale image of a digit.

The spatial pooler is designed to accept binary input to perform its function.

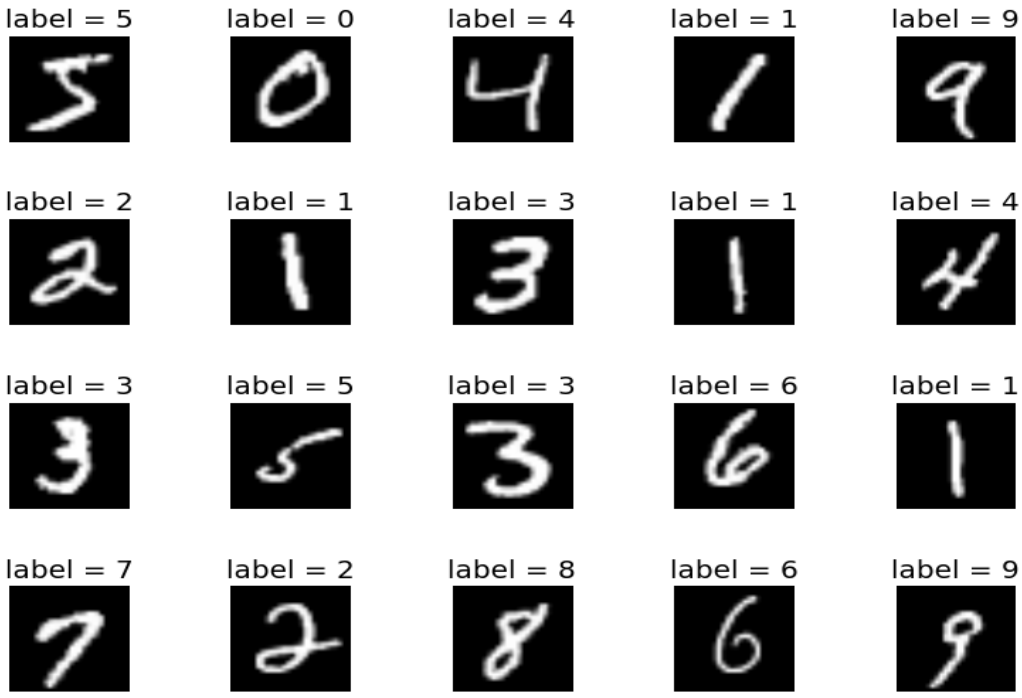

Figure 4.1: Samples of MNIST dataset and their labels. [12]

Therefore, before feeding in the input images to the spatial pooler the MNIST grayscale images [12] are first pre-processed to convert them to binary images. Thresholding technique is used to perform grayscale to binary image conversion. A threshold value of 127 is set and each pixel in the image is compared with the threshold. All the pixels with values lesser than the threshold are set to '0' and the pixels with values greater than or equal to threshold value are set to '1'. The 28 x 28 pixel values are then converted into a one dimensional vector of 784 bits of input image data. The resulting value is used as the input data for the spatial pooler. Image processing is performed on MATLAB. This step is not considered as the part of the spatial pooler hardware realization.

4.1.2 Module Verification

The pre-processed MNIST dataset images are saved in a text file and sent to the spatial pooler architecture through VHDL testbench by behavioral simulation of RTL. The RTL testbench instantiates the HTM SP Design Under Test (DUT) and reads input patterns from a text file to the HTM SP. The HTM SP SDR ouput produced is written as a separate text file. The input binary image patterns are read in sequentially by the testbench, and the process waits till the output for the given input is produced, to read the next input image data. This process continues until the specified number of training data samples and testing data samples have been read into the spatial pooler DUT.

Upon completion of all the input image data samples processing, the classification accuracy for the spatial pooler is obtained. As the algorithm follows unsupervised learning, only the image data is fed in as input to the SP and the lables are never given to the network. In this work, the HTM SP uses 50,000 MNIST training data samples and 10,000 MNIST testing data samples.

4.1.2.1 HTM SP Region Training

Spatial pooler training involves learning the input patterns to produce SDR for the given input. Learning involves updating the permanence values for the proximal synapses of the columns. Each time a new pattern is fed in to the spatial pooler, the permanence values are adjusted to produce a generalized SDR for that pattern and the spatial pooler is considered to have learned the given pattern. Since the permanence values are set to values closer to the synapse threshold value during spatial pooler initialization, it learns faster as it takes only a few training images for the region to

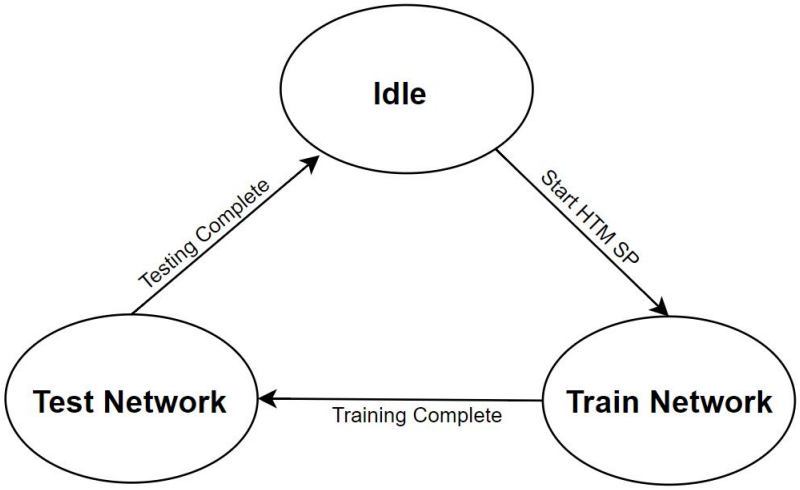

Figure 4.2: State machine represents training and testing phases of the digital design. The spatial pooler remains idle until the start signal is asserted and the spatial pooler training begins. Learning remains enabled throughout and all the train data images are fed in to learn the images. When all the train images are processed, training is complete and the learning is disabled. Now the spatial pooler is tested for test data images and the process continues until all the test data images are fed in.

start producing the right SDRs.

4.1.2.2 HTM SP Region Testing

Spatial pooler training is followed by testing to determine how the trained columns respond to the unseen input patterns or images. In this process, the learning module is disabled, therefore, testing doesn't involve updating permanence values of the proximal synapses of the columns for a given input image. Instead, the latest updated permanence values of the proximal synapses of the columns obtained during training are used to produce SDRs for the test data images.

4.1.3 HTM SP Classification Accuracy

After training and testing the spatial pooler, the next step is to calculate the classification accuracy of the spatial pooler design. Support Vector Machine (SVM) is

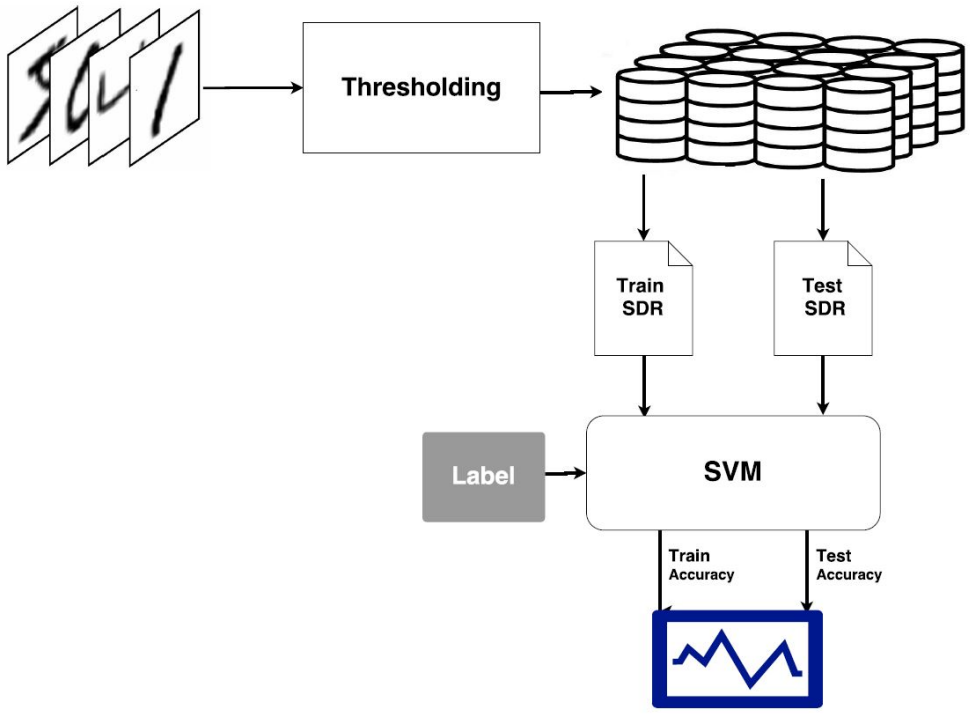

Figure 4.3: Graphical representation of design verification flow of the spatial pooler. MNIST data is processed to perform image thresholding. After the image process, the data is sent to the spatial pooler module to obtain its SDR. The spatial pooler output is sent to support vector machine to obtain the classification accuracy of the spatial pooler using MNIST label for the input data.

used to calculate the classification accuracy. Linear Support Vector Clustering (linear SVC) is performed on Python using *sklearn* library package. Linear SVC is a clustering algorithm which provides improvement to SVM. The SDR output saved in the text files generated by the spatial pooler module and MNIST database labels are used as the input parameters by the SVM to calculate the classification accuracy for both training data and testing data.

4.2 Simulations and Analysis

The scalable RTL design for the spatial pooler is trained for different number of columns and proximal synapses. User defined constant parameter values are varied

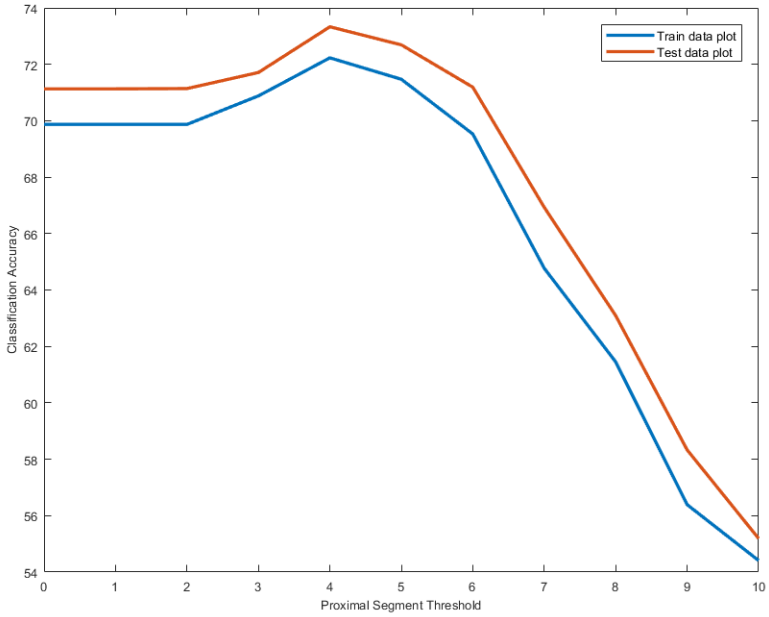

Figure 4.4: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 100 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10.

to observe how the values affect the spatial pooler classification accuracy on the digital design and to determine the best parameter values to consider for the digital design implementation.

Simulations were run for the design to determine the optimum value for the segment threshold at which the spatial pooler performs the best. The proximal segment threshold value was varied between 0 to 10 while keeping the other parameter values fixed. The spatial pooler classification accuracy varied inversely with respect to the segment threshold value. The drop in the performance for the higher threshold values is due to information loss. For the lower threshold values only noise is removed but as the threshold value is increased, the information is also lost hampering the classification accuracy. The segment threshold values were varied for varying number of columns and the variations for them showed the similar pattern.

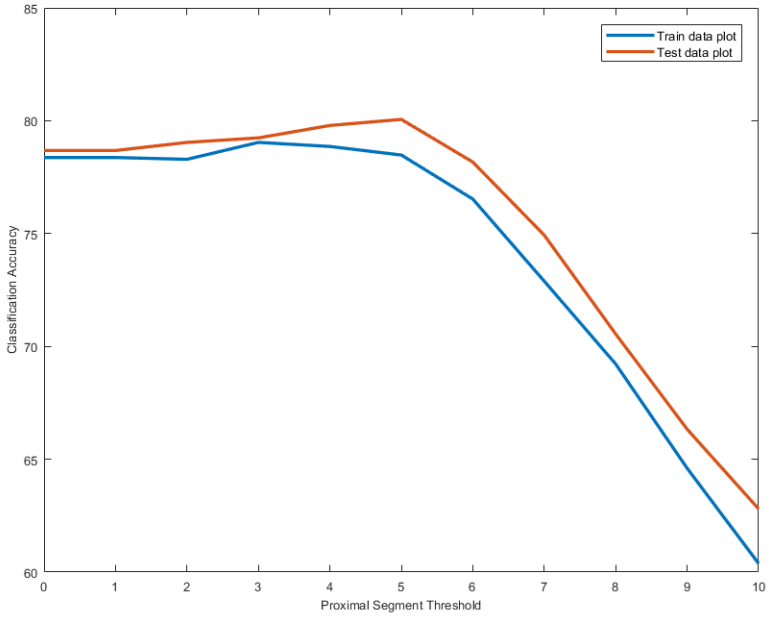

Figure 4.5: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 200 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10.

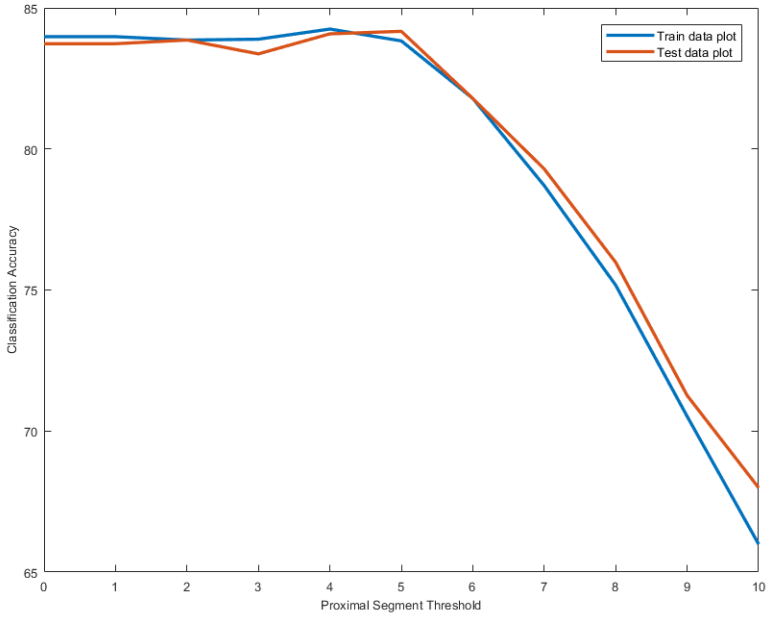

Figure 4.6: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 400 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10.

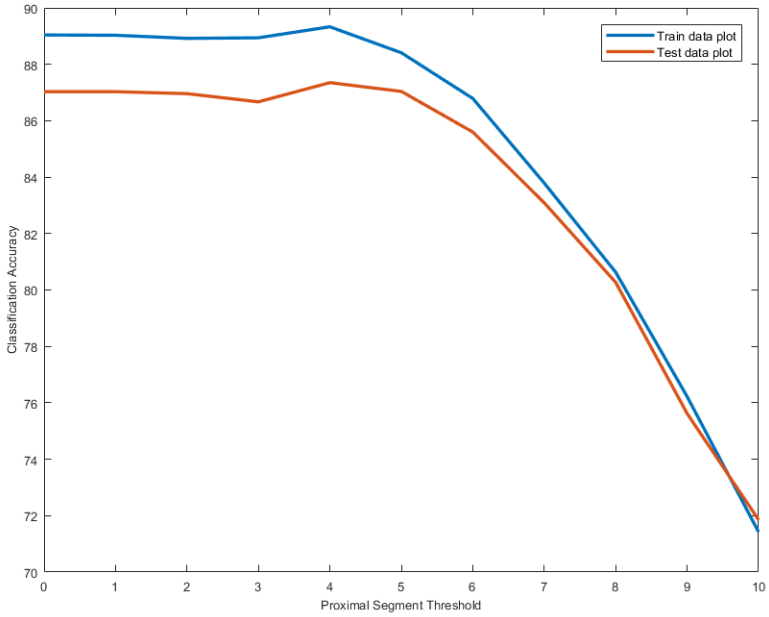

Figure 4.7: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 784 columns and 100 proximal synapses per column and varying the proximal segment threshold between 1 and 10.

A series of simulations were performed to determine the percentage of winning columns to be considered for the digital design of the spatial pooler. The percentage of winning columns was varied while keeping other parameter values fixed. The spatial pooler performed better for the regions with more than 10% of its columns marked as the winners. To maintain the sparsity and reduced complexity, the spatial pooler must have a smaller percentage of winning columns. So, the smallest percentage at which the spatial pooler started providing better performance was chosen as the desired parameter in this design.

Finally, the number of columns in the spatial pooler region were varied while keeping all the other parameter values at their optimum values. The spatial pooler was tested for varying number of columns between 100, 200, 400, and 784. The spatial pooler classification accuracy is linearly proportional to the number of columns.

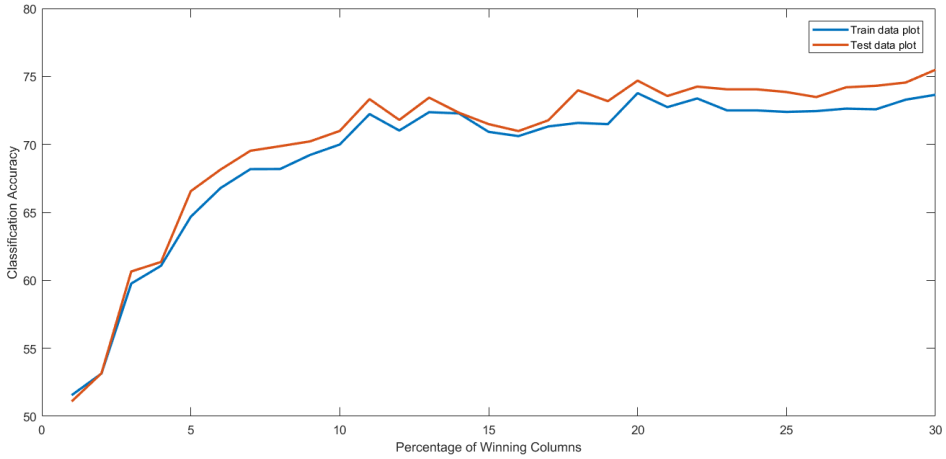

Figure 4.8: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 100 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30.

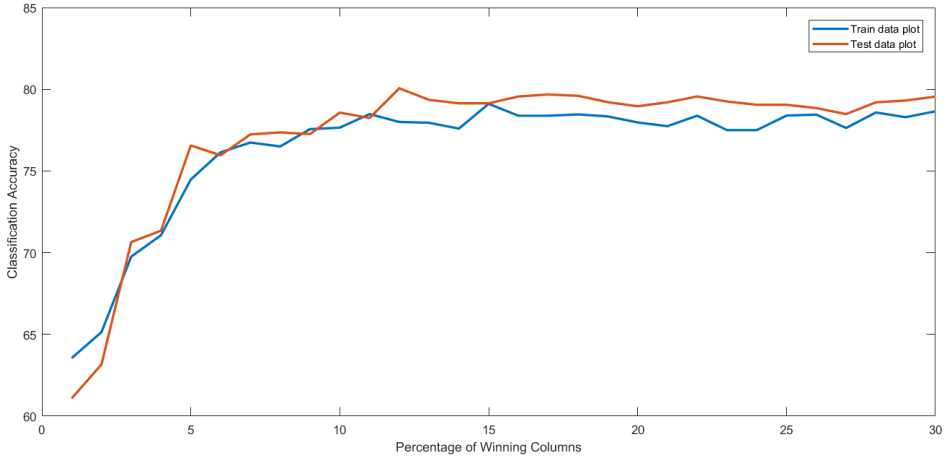

Figure 4.9: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 200 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30.

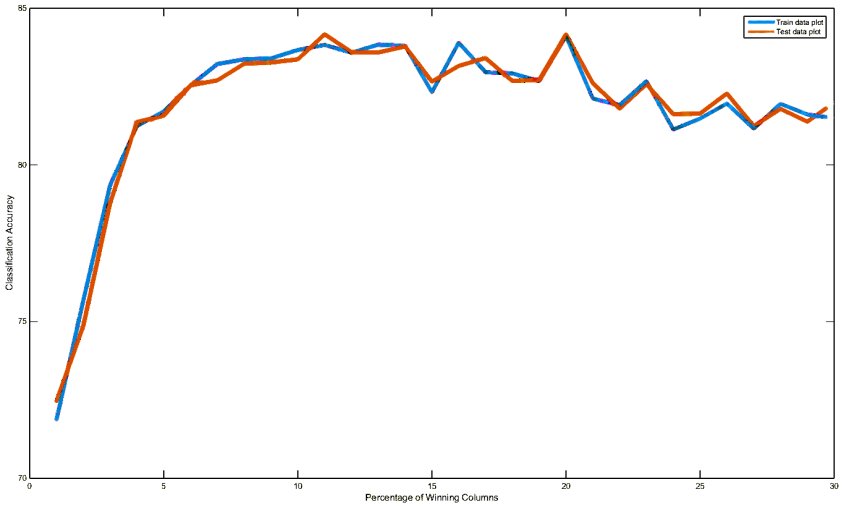

Figure 4.10: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 400 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30.

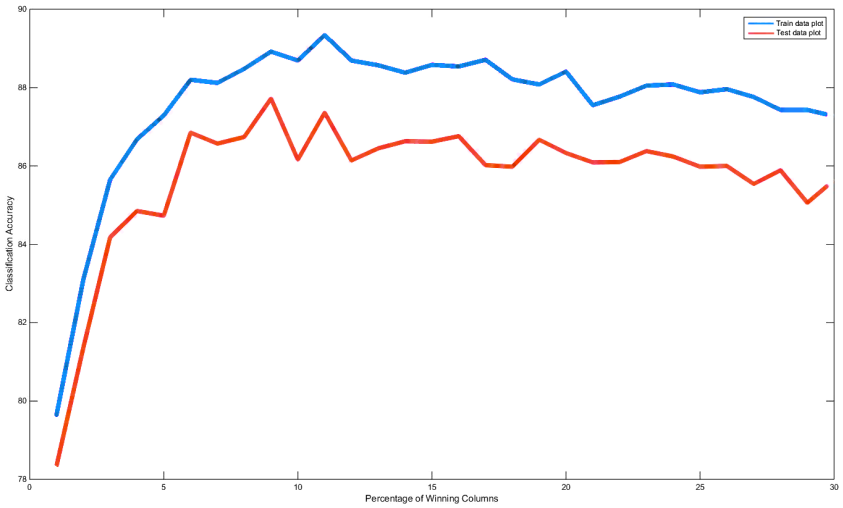

Figure 4.11: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a region with 784 columns and 100 proximal synapses per column and varying the percentage of winning columns between 1 and 30.

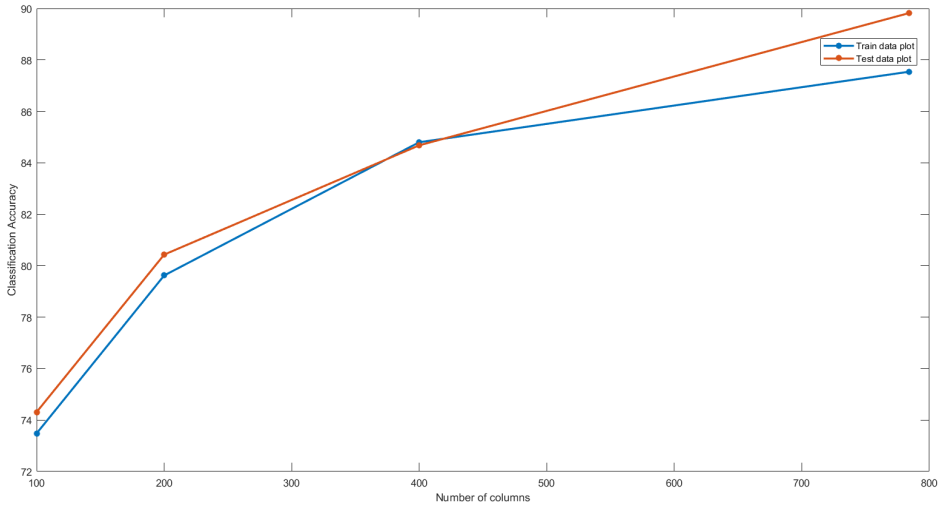

Figure 4.12: Plots showing classification accuracy of the spatial pooler design for MNIST train data and test data sample for a regions with 100, 200, 400 and 784 columns and 100 proximal synapses per column, keeping their parameter values fixed at their optimum values.

Simulation time for the design increases with the increase in the number of columns. Classification accuracy of upto 89.34% on training dataset and 87.36% on testing dataset was achieved. The accuracy could be further increased by increasing the number of columns.

Figures 4.13 and 4.14 explain the variation in the classification accuracy in the presence of noise for both train data and test data. Different types of noise was deliberately added to the MNIST dataset to check how the noisy input patterns have negative impact on the classification accuracy of the SP digital design. It was seen that the accuracy reduced with the increase in the density of noise in the image due to the degraded representation of the pattern. However, the results show that the system can handle noise with a density of 20%.

Considering area requirement and hardware resource utilization, Spatial pooler with 400 columns with 100 synapses per column was considered to synthesize for

ConstantParameter	*Parameter Value*
Clock Frequency	100 MHz
Proximal synapse threshold	128
Proximal segment threshold	5
Winning columns	11%
permanence increment	1
permanence decrement	2
$c_p update$	3
Minimum permanence	0
Maximum permanence	256

Table 4.1: User defined constant vales for the spatial pooler parameters.

Virtex7 FPGA.

4.2.1 Synthesized Model Device Utilization

The spatial pooler design architectures are synthesized for Virtex7 XC7VX330T FPGA. The size of the spatial pooler was kept fixed at 400 columns and 100 synapses per column for synthesis. The design was run at the 100 MHZ clock frequency. The area requirement for both design architectures are estimated. The device utilization of the conventional architecture on Virtex7 FPGA was lesser than the parallel architecture. Tables 4.2 and 4.3 show the area requirements for the architectures obtained after synthesis. While area requirement is a tradeoff for parallel architecture, speedup can be achieved by using the parallel architecture. The following sections compare the two architectures in terms of power and speed.

4.2.1.1 Execution Time

The execution times for both the architectures were compared. Since initialization process is same for both the architectures, the execution time period for initialization remains equal. The execution time for a single train data sample decreased by 40% in

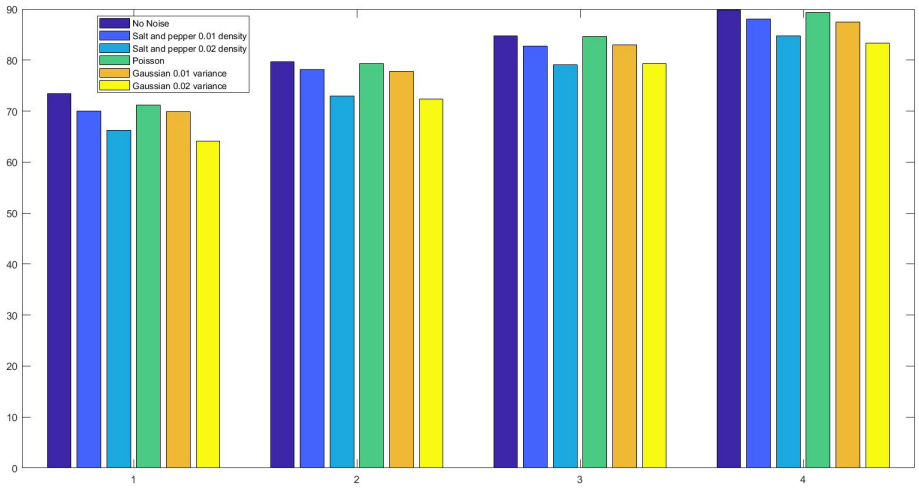

Figure 4.13: Plots showing classification accuracy of the spatial pooler design for MNIST train dataset with noise. 1 represents the performance for spatial pooler with 100 columns; 2 represents the performance for spatial pooler with 200 columns; 3 represents the performance for spatial pooler with 400 columns; 4 represents the performance for spatial pooler with 784 columns.

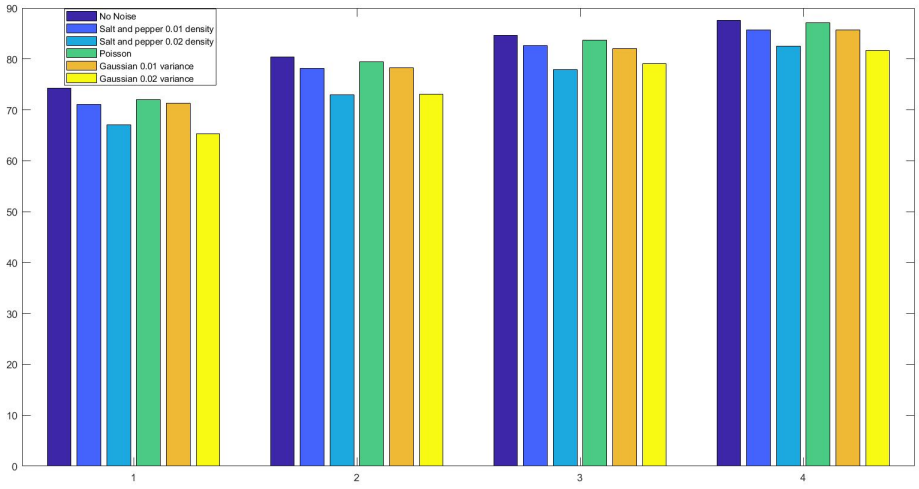

Figure 4.14: Plots showing classification accuracy of the spatial pooler design for MNIST test dataset with noise. 1 represents the performance for spatial pooler with 100 columns; 2 represents the performance for spatial pooler with 200 columns; 3 represents the performance for spatial pooler with 400 columns; 4 represents the performance for spatial pooler with 784 columns.

Conventional mHTM SP	*Used Resources*	*Available Resources*	*Utilization*
Number of Slice Registers	3832	408000	1%
Number of Slice LUTs	32015	204000	15%
Number of occupied Slices	9577	51000	18%
Number of LUT FF pairs used	32618	-	-
Number of Bonded IOBs	405	600	67%
Number of RAMB36E1	18	750	2%
Number of RAMB18E1	1	1500	1%
Number of BUFG	1	32	3%
Number of OLOGICE2S	400	-	-

Table 4.2: Area estimates of Conventional mHTM SP architecture on a Xilinx Virtex7 XC7VX330T target device.

Parallel mHTM SP	*Used Resources*	*Available Resources*	*Utilization*
Number of Slice Registers	4610	408000	1%
Number of Slice LUTs	38296	204000	18%
Number of occupied Slices	10290	51000	20%
Number of LUT FF pairs used	39016	-	-
Number of Bonded IOBs	405	600	67%
Number of RAMB36E1	0	750	2%
Number of RAMB18E1	1	1500	1%
Number of BUFG	1	32	3%
Number of OLOGICE2S	400	-	-

Table 4.3: Area estimates of Parallel mHTM SP architecture on a Xilinx Virtex7 XC7VX330T target device.

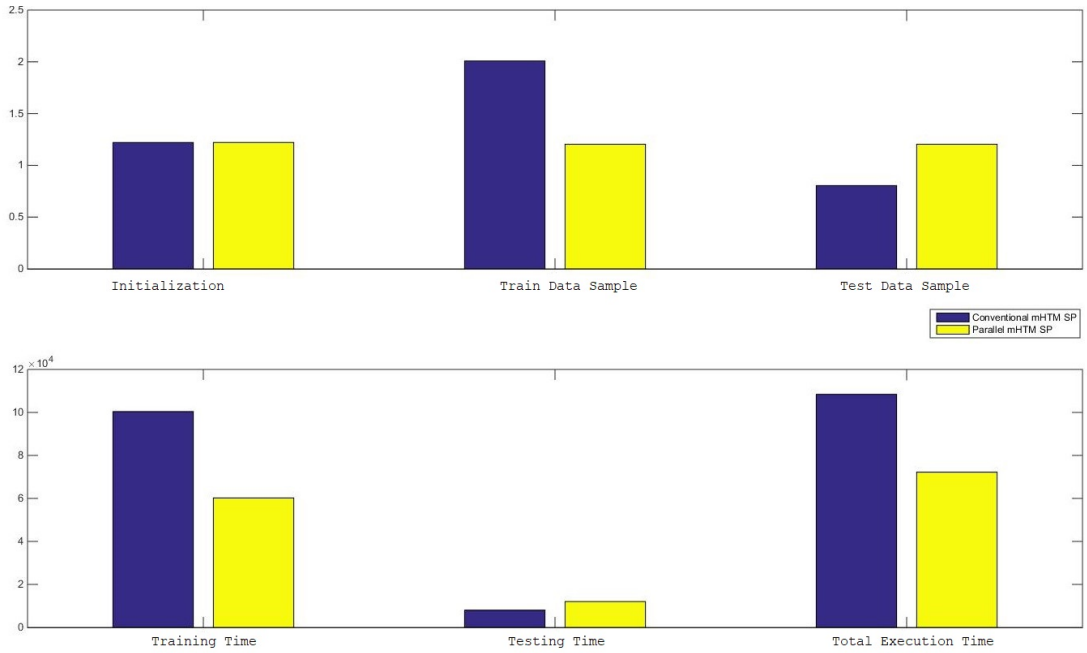

Figure 4.15: Plots comparing the execution time for the Convention mHTM SP and Parallel mHTM SP architectures and their speedup.

the parallel architecture due to parallel computation of overlap and learning modules. However, the execution time for the test data sample increased by 50% due to decision states that remain idle when learning is disabled. The overall speed up achieved by the parallel architecture is 30%.

Execution Time	*Conventional mHTM SP (ms)*	*Parallel mHTM SP (ms)*
Network Training	100405.5	60205.5
Network Testing	8041	12041
SP Initialization	1.220	1.220
Per Train data sample	2.008	1.204
Per Test data sample	0.804	1.204
mHTM SP	108447	72249

Table 4.4: Execution timing comparison for the Conventional and Parallel mHTM SP architectures on a Xilinx Virtex7 XC7VX330T target device.

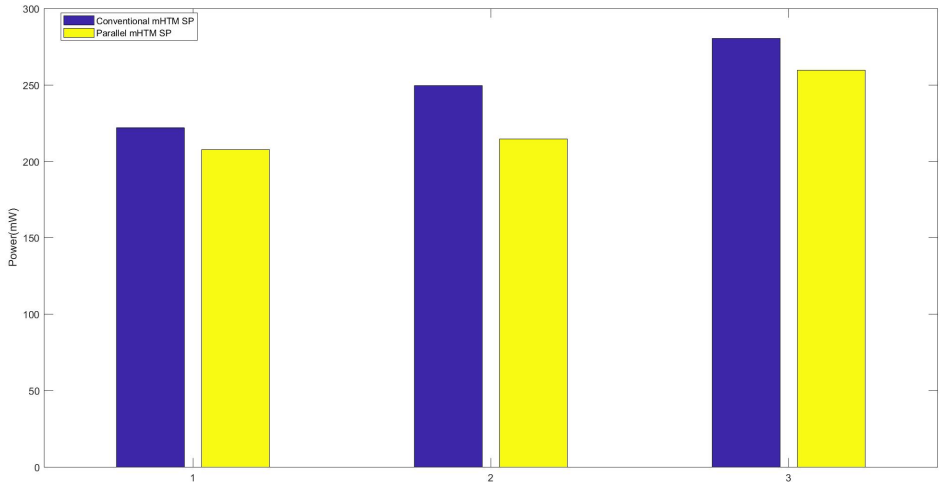

Figure 4.16: Plots comparing total power consumption for Conventional and Parallel mHTM SP architectures on Xilinx Virtex7 XC7VX330T target device. 1 indicates power consumption for the spatial pooler with 100 columns. 2 indicates power consumption for the spatial pooler with 200 columns. 3 indicates power consumption for the spatial pooler with 400 columns.

Architecture	*Total Power (W)*	*Dynamic (W)*	*Static (W)*
Conventional mHTM SP	0.280	0.102	0.178
Parallel mHTM SP	0.259	0.081	0.177

Table 4.5: Power estimates of Conventional and Parallel mHTM SP architectures on a Xilinx Virtex7 XC7VX330T target device.

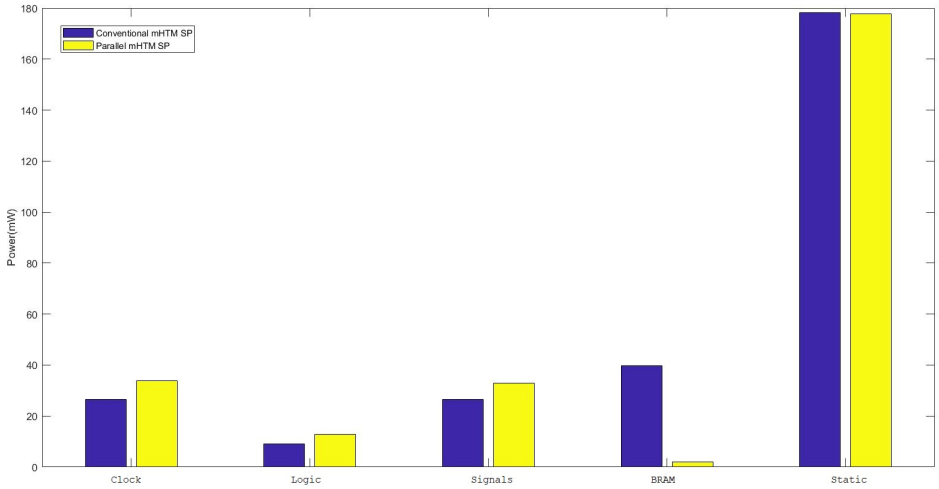

Figure 4.17: Plots comparing individual component power consumption on for the Conventional mHTM SP and Parallel mHTM SP Architectures with 400 columns and 100 synapses per column on Virtex 7 FPGAs

4.2.1.2 Power Analysis

Power analysis was performed on the architectures using XPower Analyzer. Both the architecture designs have static power consumption of about 178 mW. The dynamic power vary significantly. The parallel architecture consumes lesser BRAM power due to reduced read and write accesses to the memory. Thus, the total power consumption for the parallel architecture is lesser.

Chapter 5

Conclusion

5.1 Conclusion

This thesis work presents the digital design of the mathematical model of the Hierarchical Temporal Memory spatial pooler. This unsupervised, online machine learning algorithm is successful in mapping the spatio-temporal data from the input domain to generalized sparse distribution representations. The SDR produced by the spatial pooler are used by the HTM temporal memory for making predictions.

In this work, two architectures for the spatial pooler are implemented. Parallelism was exploited to a certain degree within the design modules. The design performance is evaluated against MNIST dataset using SVM. Results show that upto 89.34% training accuracy and 87.36% testing accuracy can be achieved for the larger architecture size. However, the designs are synthesized for smaller number columns for Xilinx Virtex7 XC7VX330T FPGA device to overcome area overhead. Different parameter values of the spatial pooler are varied to demonstrate the algorithm variations with the change in those parameter values. Area and power consumption for the design implementations are analysed.

Chapter 6

Future Work

This chapter describes in brief some of the several aspects that needs to be explored in the future while designing the mathematical model of the HTM SP on FPGA. One of these include increasing parallelism in the design. Parallelism was achieved at the modular level in this work. However parallelism can be extended further at columnar level. There are many ways to improve the throughput in the design architecture, some of which include having multiple module instances operating in parallel.

Dividing Block RAM into smaller blocks can result in increased throughput of the design. The design performance could be further increased by introducing multiple clock domains in the system. Switching to an external non-volatile memory from the Block RAM would result in power optimized design.

Future work also includes development of a hardware implementation of Temporal Memory that supports the proposed spatial pooler and matches its throughput on hardware to visualize the efficiency of the Hierarchical Temporal Memory overall.

Bibliography

[1] J. Mnatzaganian, E. Fokoué, and D. Kudithipudi. *"A Mathematical Formalization of Hierarchical Temporal Memory Cortical Learning Algorithm's Spatial Pooler,"* arXiv preprint arXiv:1601.06116, 2016.

[2] D. Maltoni *"Pattern recognition by hierarchical temporal memory,"* DEIS Univ. Bologna, Tech. Rep., pp. 1-46, Apr. 13, 2011.

[3] P. Gabrielsson, R. Konig, and U. Johansson, *"Evolving hierarchical temporal memory-based trading models,"* in EvoApplications 2013-Applications of Evolutionary Computing, Vienna, April 3-5, 2013.

[4] Y. Cui, C. Surpur, S. Ahmad, and J. Hawkins, *"Continuous online sequence learning with an unsupervised neural network model,"* arXiv preprint arXiv:1512.05463, 2015.

[5] S. Lattner, *"Hierarchical temporal memory-investigations, ideas, and experiments,"* Master's thesis, Johannes Kepler Universitat, 2014.

[6] M. Leake, L. Xia, K. Rocki, and W. Imaino, *"A probabilistic view of the spatial pooler in hierarchical temporal memory,"* World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering, vol. 9, no. 5, pp. 1111–1118, 2015.

[7] Jeff Hawkins and Sandra Blakeslee. *On Intelligience.* Macmillan, 2007.

[8] J. Hawkins and D. George. *Directed behavior using a hierarchical temporal memory based system.* Available at http://appft1.uspto.gov/netacgi/nph-Parser?

Sect1=PTO1Sect2=HITOFFd=PG01p=1u=/netahtml/PTO/srchnum.html r=1f=Gl=50s1=20070192268.PGNR.

[9] Dileep George. *How the brain might work: A hierarchical and temporal model for learning and recognition.* PhD thesis, Stanford University, 2008.

[10] *"Hierarchical temporal memory including htm cortical learning algorithms,"* Available at http://numenta.com/assets/pdf/whitepapers/hierarchical-temporal-memory-cortical-learning-algorithm-0.2.1-en.pdf, 2011, accessed on 2014-11-02.

[11] [LeCun et al., 1998a] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998. [on-line version]

[12] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010

[13] Numenta Inc. Hierarchical Temporal Memory: Concepts, Theory, and Terminology. Tech. rep. 791 Middlefield Road Redwood City, CA 94063: Numenta Inc., Mar. 2006.

[14] Dileep George. "How The Brain Might Work: A Hierarchical and Temporal Model for Learning and Recognition." PhD thesis. 450 Serra Mall Stanford, CA 94305: Stanford University, June 2008.

[15] Ryan William Price. "Hierarchical Temporal Memory Cortical Learning Algorithm for Pattern Recognition on Multi-core Architectures." MA thesis. 1825 SW Broadway, Portland, OR 97201: Portland State University, Jan. 2011.

[16] Xi Zhou and Yaoyao Luo. "Implementation of Hierarchical Temporal Memory on a Many-Core Architecture." MA thesis. PO Box 823 SE-301 18 Halmstad, Sweden: Halmstad University, Dec. 2012.

[17] Abdullah M. Zyarah. "Design and Analysis of a Reconfigurable Hierarchical Temporal Memory Architecture." MA thesis. 1 Lomb Memorial Dr, Rochester, NY 14623: Rochester Institute of Technology, June 2015

[18] Lennard Streat, Dhireesha Kudithipudi and Kevin Gomez. Non-volatile Hierarchical Temporal Memory: Hardware for Spatial Pooling, 2016; arXiv:1611.02792.